



Complexity of In-Code Variability: Emergence of Detachable Decorators

Jakub Perdek^(✉)  and Valentino Vranić 

Institute of Informatics, Information Systems and Software Engineering Faculty of
Informatics and Information Technologies, Slovak University of Technology in
Bratislava, Bratislava, Slovakia
{jakub.perdek,vranic}@stuba.sk

Abstract. This paper presents a study on how selected approaches to expressing variability in code affect code complexity. To evaluate and compare the complexity of essential aspects of different approaches to in-code variability management, we designed five prominent cases of how variability is expressed in code: using decorators, using decorators without variability configuration expressions, using wrappers, using decorators with additional unwanted dead code constructs not being included for illegal decorators, and with no variability expressed in code at all. To measure code complexity in these cases, we used a framework for evaluating TypeScript code that we implemented in Java. Our framework is capable of assessing 15 metrics, comprising several variants of the LOC, Halstead, cyclomatic complexity, and cyclomatic density metrics. Decorators are detachable because they are decorating particular code construct with a predefined naming convention and no effects on code. The study was conducted on a software product line aimed at graphical applications we developed for evaluation purposes. We came to a range of interesting findings, such as that the detachable decorator version of introduced annotations is significantly less complex than other ways of expressing variability in code, annotations in comments (as in `pure::variants`), and tags (as in frame technology) do not directly affect the complexity of business functionality, decorators can be entirely separated from business logic, etc.

Keywords: code complexity · software product lines · variability management · decorators · variability configuration · comparative analysis

1 Introduction

Variability management is the central part of software product lines. Its in-code realization may significantly increase code complexity [5, 9, 24]. The configuration of products in software product lines is commonly managed with transformations or with visual languages [24]. Consequently, the exponential increase [27] of variants by the number of features complicates manual corrections [3], even when

resolving conflicts while incorporating new features into an existing software product line [1]. These corrections and conflicts should be resolved automatically. However, the complexity of in-code variability management constructs is neither evaluated, nor automatically applied to optimize configuration expressions. Its because they are used in comments or configuration variables are interleaved with business logic code.

This paper presents a study on how selected approaches to expressing variability in code affect code complexity. It is organized as follows. Section 2 presents some approaches to dealing with the complexity of expressing variability in code. Section 3 explains how we designed our study on how selected approaches to expressing variability in code affect code complexity, Sect. 4 explains how the study was performed and presents its results. Section 5 discusses the results. Section 6 relates this study to what others have done. Section 7 concludes the paper.

2 Variability Management Constructs and Code Complexity

Variability management is preserved in code mainly through wrapped comments such as in `pure::variants` [23]. An example is shown in Listing 1.1. Code fragments that implement variable features are being marked using annotations manually or in an automated fashion to allow their tracing and composition [5].

In addition to comments, tags can be used to express code templates as in the frame technology [17, 18]. This requires dedicated transformation tools to be applied before the actual compilation. Another option to manage variability in code is to use conditional compilation [6], which also relies on tags (preprocessor directives). However, these approaches lead to polluting the code with tags [24], making it more complex.

```
1 //PV:IFCOND(pv:hasFeature(HazardWarning))
2 static int warning_lights_value; [REMAINING CODE OF HazardWarning...]
3 //PV:ENDCOND
```

Listing 1.1. Expressing in-code variability in `pure::variants` (adopted from `pure::systems` [23]).

Variability management based on aspect-oriented programming to some extent resolved evolvability, modularity, and code pollution issues [6], but remained not fully supported. Furthermore, aspects leave the affected code oblivious of its effects [7]. Variability configuring variables scattered through various aspects are usually used and mixed with the rest of the business code.

Annotations are usually realized with comments which require additional preprocessing tools [5]. It's because of the need for an easy establishment of annotation-based. [5] software product lines, technology support, and the mentioned restrictions. The code fragments that implement variable features are wrapped by predefined comments that are later processed to allow for their tracing.

Complexity metrics have been successfully applied to measure the difficulty of code comprehension [11]. However, the comments usually do not contribute to this process because they are omitted during compilation. Under such circumstances, evaluating the complexity of these comments and related quality metrics is impossible. If aspect-oriented programming is used, conditions can be expressed directly by variables in aspects. In such case, separating and measuring the complexity of particular constructs is challenging because variables that guard variability conditions according to configuration tend to be mixed with business functionality.

Configuration expressions are formulas used to decide whether to include or exclude processed code fragments if conditions are fulfilled. They can contain additional information associated with an annotated variation point. Analysis of the impact of their hierarchically expressed version [21] is required for flexible configuration in a particular context. An example of such an expression is in Listing 1.4.

```
1 // @ts-ignore
2 @DecorSRVC.skipLVP({"algoType": "[',
  A1', 'A2', 'A3']"}, "import { State
    } from '../store';") var newA;
3 //import { State } from '../store';
```

Listing 1.2. Annotated import statement by decorators.

```
1 @DecorSRVC.wBlock({"zoom": "true"})
2 public zoom(zoomHTML: any): void {
3     //DO ZOOMING
4 }
```

Listing 1.4. Annotated method by decorators.

```
1 var EXP_START6 = { "algoType": "[',
  A1', 'A2', 'A3']" };
2 import { State } from '../store';
3 var EXP_END6 = { "EXP_END": "--" };
```

Listing 1.3. Wrapped code of the import statement.

```
1 var EXP_START0 = {"zoom": "true"};
2 public zoom(zoomHTML: any): void {
3     //DO ZOOMING
4 }
5 var EXP_END0 = { "EXP_END": "--" };
```

Listing 1.5. Wrapped code of the method.

3 Designing the Study

In the study reported in this paper, the code complexity evaluation of essential aspects of variability management was performed on adaptation of annotations in comments—as they are used in pure::variants—, preprocessor directives—as they are used in conditional compilation—, and tags—as they are used in frame technology— with the code construct equivalents we proposed. The type and use of annotations are taken from our method of developing lightweight aspect-oriented software product lines with automated product derivation [21]. Similarly, we consider the complexity of used configuration expressions and overhead caused by necessary dead code. We adapted programming language implementations of decorator pattern to design decorator-bound form which is perceived as

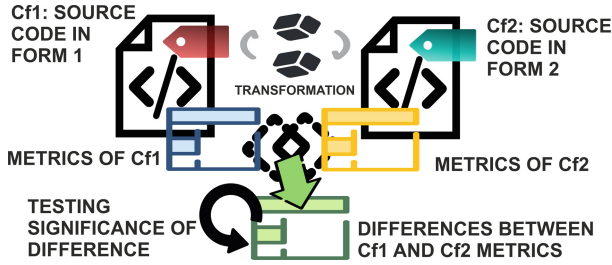


Fig. 1. Comparative analysis of code constructs complexities of variability management.

the cleanest and most adaptable to us. The possibility of transforming it into different variants enables flexible adaptation of conventional versions of annotations in source-code comments. This form with unsupported decorators is converted into a compilable version with supported constructs that can be evaluated and used further in its less concise structure. In TypeScript this is caused by unavailable types of decorators or illegal decorators [28]. Many tools cannot process the latter, but these constructs remain in the generated abstract syntax trees. Associated configuration expressions can be optionally excluded to evaluate their complexity in a particular context and improve them accordingly.

All mentioned challenges with measuring and comparing code complexity are solved with our framework. Specifically, the code complexity from each file is transformed into a pair of variability management versions for analysis and visualized with code itself, evaluated complexities, differences of evaluated complexities, and a statistic test performed to measure if this pair of variability management versions/cases significantly differ. We propose a process to compare two annotated files with variability management fully realized in code as shown in Fig. 1. This process is supported by a framework, which is available on Github,¹ along with all scripts, software product line aimed at graphical applications we developed for evaluation purposes, and resulting artifacts.

3.1 Adapting the Wrappers and Designing the Detachable Decorators

We designed wrappers equivalent to available variability management constructs based on conditional compilation constructs, variability comments as in pure::variants, and tags as in the frame technology (either pure or supported by aspect-oriented programming). Our wrappers are a part of the syntax of the programming language being used for software product line development (TypeScript in our case). They are active code constructs, i.e., they are not preprocessor directives, tags, nor comments.

¹ <https://github.com/jperdek/variabilityMgmtCodeConstructsComplexity>.

Wrapping happens for the array of members or statements in the final abstract syntax tree by putting variability management elements before and after each selected sequence of members or statements. Additionally, an alternative “else” branch can optionally extend the wrapped functionality and help handle situations where given features are unavailable for particular configurations. Compared to decorators, nested wrappers are usually more difficult to read.

Contrary to this, we are introducing an adjustment of decorators for variability management taking into account their limited support to annotate only function parameters, classes, class methods, and variables [2]. Only these annotable variable units at the code level are prioritized in representing variability. Accordingly, decorators are proposed as a more adaptable form for managing variability and evaluating complexity. They can be associated with modular source code structures in a more comprehensible way. Similarly, they affect code execution only in cases of providing support for dynamic variability management. They are fully distinguished from the rest of the business logic (its constructs) by the predefined names, which makes them detachable and independent of variability management. Compared to decorators, the wrappers is highly applicable, but cannot be restricted or bound to particular code structures. The difference can be seen by comparing Listing 1.2 to Listing 1.3 and Listing 1.4 to Listing 1.5. In all listings, the same business logic is denoted as variability. Specifically, the import statement occurs in the first pair and the zoom method in the second. The first member of the pair consists of a decorator beginning with @, and the second is a conventional wrapper bounded with two initialized variables. Listings 1.2 and 1.4 employ a decorator beginning with @, while the other two listings employ a conventional wrapper bounded with two initialized variables.

Other forms can be based on constructs, such as if statements or for loops, but their use directly interleaves with code and possibly negatively affects modularity.

Each of the mentioned forms needs developers to be aware of the product derivation mechanism to the depth necessary to know its capabilities. For example, when code is annotated in wrong places, expressed with an unknown character sequence, or in the case of templates, it causes additional changes to the previous development style.

A particular product derivation mechanism usually does not demand preserving code modularity and extendability when variability needs to be handled at the code level. Consequently, if recognizable constructs for this mechanism are not preserved in source code, the mechanism cannot manage and synchronize how available code constructs are used. Extending their semantics to variability management increases overall complexity, especially if conditional compilation is used. In particular, the statements that express variability cannot be easily semantically distinguished from those statements that express business logic.

To evaluate and compare the complexity of essential aspects of different approaches to in-code variability management, we designed the following cases:

Case 1. Variability is expressed using detachable decorators

- Case 2.** Variability is expressed using detachable decorators, but without variability configuration expressions
- Case 3.** Variability is expressed using wrappers
- Case 4.** Variability is not expressed at all
- Case 5.** Variability is expressed using detachable decorators, but additional unwanted dead code constructs are not included for illegal decorators

Each case essentially differs in one or more characteristics, including the availability and format of configuration expressions as inner members (JSON or attributes), the way code constructs are used for variability management (wrapping, annotating, or none), the type of used constructs for variability management (decorators, variables, or preprocessor directives), and the necessity to use dead code for visual adaptation of variability management code constructs to places affected by variability. We did not consider conditional compilation with associated preprocessor directives because such representation interleaves with business logic code. We analyzed only configuration expressions in the JSON format due to the possibility of directly analyzing them as JavaScript/TypeScript objects, modeling and preserving hierarchic relations with feature models in code, and using a more concise format over conditions consisting of variables that usually interleave with business logic code.

Despite the mentioned characteristics, we must also consider available code constructs in a given programming language, capabilities of code complexity evaluation tools, and options to position available code constructs near target places visually. The comparative analysis proposed here was realized in scenarios each of which compares two of the in-code variability management cases we designed.

3.2 Hypotheses and the Process

We propose the following hypotheses to evaluate the effect of in-code variability constructs on code complexity, namely cyclomatic complexity, LOC, and Halstead measures:

- Hypothesis 1.** Variability expressions extracted from annotations do not significantly change the complexities of most evaluated metrics.
- Hypothesis 2.** Changing from wrappers to detachable decorators significantly improves the complexity of most evaluated complexity metrics.
- Hypothesis 3.** Removal of all variability constructs from Case 1 does not significantly change at least one of the evaluated complexity metrics.
- Hypothesis 4.** Unwanted dead code constructs significantly change complexity measured by most evaluated complexity metrics.

Validating Hypothesis 1 involves evaluating the complexity of configuration expressions represented in JSON format [21] with all mentioned metrics to show how code complexity is increased in a particular context. Specifically, Case 1 and 2 are compared. We assume that the complexity of configuration expressions significantly affects particular complexity metrics. Minor optimizations to

complexity evaluation can be performed by putting JSON into a string so that it forms one expression and measuring the final effect. This change will comprise the isolation of variability configuration expressions during measurements from the rest of the code, especially from business logic.

In future, such expressions could be collected, updated, compared with other expressions, and possibly optimized in an automated process. Making configuration expressions less complex and more comprehensible can help developers quickly learn relations from automatically generated semantic structural views [22] of an annotation-based software product line [5] (for example, with our matrix-based approach to structural and semantic analysis in software product line evolution [22]).

Configuration expressions influence the code only marginally if these differences are insignificant. Finally, the measured effect can be used further to design more optimal expressions to suit various extrafunctional requirements in an automated way.

Validating Hypothesis 2 involves evaluating the code complexity measures of the wrappers (based on already used wrappers in the form of comments) with detachable decorators (modern decorators) for variability management. Some of their negative effects on code complexity are already known [21]. Firstly, the complexity is increased in two places simultaneously when the content is bounded. In an abstract syntax tree of a TypeScript program, wrapping a particular element is possible only by adding neighbors before and after in an array of statements or members. Secondly, the nested code is hard to read when code-variability constructs must be paired. The information about how precisely quality is changed for different metrics is evaluated by comparing the complexity of wrappers (Case 3) with the one based entirely on detachable decorators (Case 1).

Validating Hypothesis 3 involves evaluating how complex variability constructs are in the particular context. In particular, we intend to find at least one metric that is significantly unaffected by variability management constructs. Such measurements are achieved by removing all variability constructs (Case 4) and comparing them to the original software product line with variability management (Case 1 or 3). The corresponding functionality can still be grouped according to used configuration expressions and assessed separately. Identification of additional complexity after introducing variability management is a basis for distinguishing how complex this functionality is.

Validating Hypothesis 4 involves evaluating the effects of legalizing some of the currently illegal detachable decorators on the code complexity of variability management. In the case of visual relatedness caused by the positioning of these variability constructs close to variation points, their syntactic representation is not connected with their visual one. Additionally, their transformation into another should be considered to evaluate approximated complexity. None of the available detachable decorators support an ideal form that allows for annotating various one-line code fragments. Even illegal detachable decorators used to annotate a particular variable require another dead code construct to be positioned near the target place to cover a variable line of code, such as an import

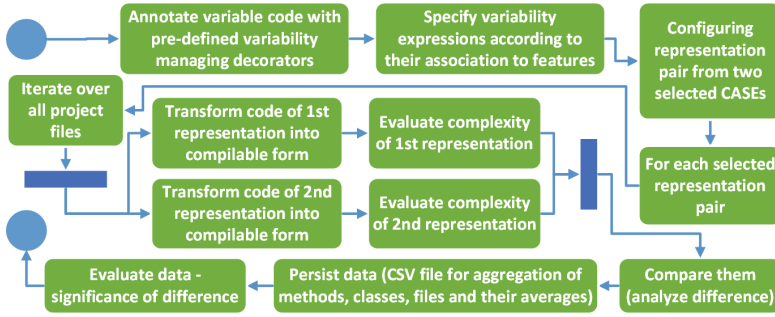


Fig. 2. Validating the hypotheses.

(see Listing 1.2) or call of a particular functionality. Transformation into the case with all supported detachable decorators (Case 1) is required to observe effects on quality, especially to source code complexity in a given context. Consequently, dead code constructs affect final complexity and can be preserved during transformation. In addition, the complexity of used detachable decorators can be approximated from average values of measurements taken previously or simply the compilable wrappers to which the code is transformed is used. We applied the latter in our study.

The process of validating the proposed hypotheses is displayed in Fig. 2. The base/actual case is transformed into a new one, followed by code complexity measurements. Finally, the subtraction of complexities determines the complexity of extended or different functionality. Firstly, variability configuration expressions are put into adapted decorator or wrapper code constructs to cover various variable code fragments. Secondly, project files are loaded and transformed into each pair of particular cases. Thirdly, the complexity is evaluated for each member with their difference in the separate measurements. Transformed scripts used to measure complexity are optionally persisted into files. Finally, data are analyzed according to chosen hypotheses and possible use cases.

4 Performing the Study

New code constructs in a particular programming language or their visual positioning in the code allow for variability management constructs to appear cleaner. For example, decorators in TypeScript can be customized with their own easily recognizable name and used to automatically transform these recognized marked points (variation points) into wrappers or even any evaluable forms. Additionally, they can be transformed into constructs that are impossible to distinguish automatically or where this process is error-prone. Their flexibility can be compared according to the code complexity of such variability markers or even associated configuration expressions used in code under a given context, such as annotating certain classes or files in a software product line as variable.

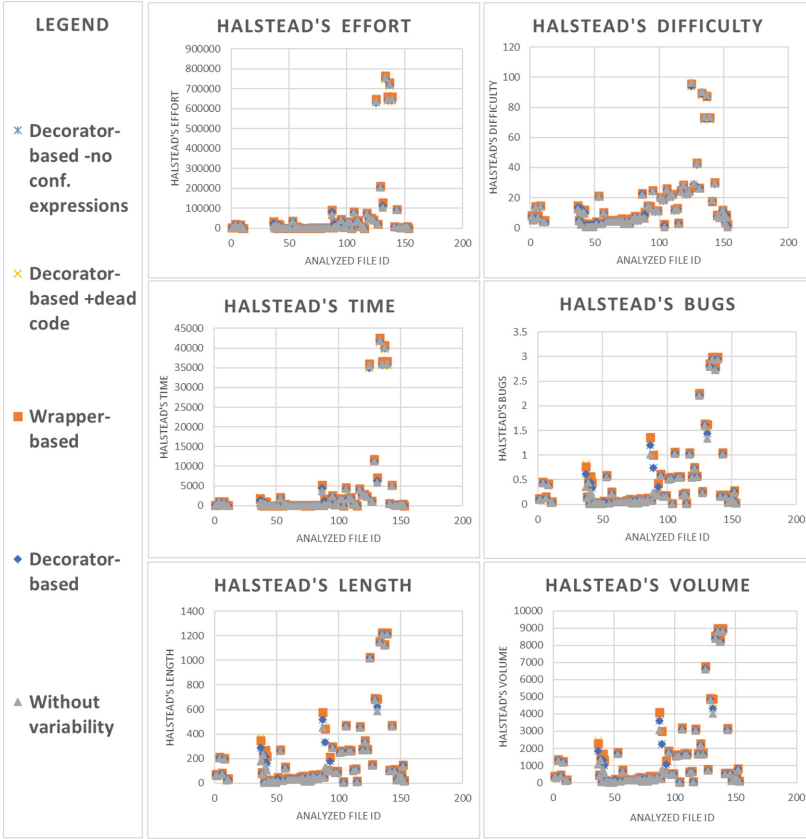


Fig. 3. Pair comparison of evaluated particular complexity metric on variability annotated files among all introduced cases: Part 1

Considering them as code constructs, their effect on the resulting code complexity can be evaluated similarly to that of a variability-unaffected code. In our study, we omit variability-unaffected files and all their code and instead focused on standalone files, their imports, classes, and various types of methods. Specifically, focus on files as standalone units is essential for considering the variability of import statements. In case merging all files together will cause not correct evaluation of overall complexity. Thanks to the TyphonJS-ESComplex service [15], source code complexity metrics are evaluated, considering decorator complexities. Supported ones are cyclomatic complexity (cyclomatic number and cyclomatic density), Halstead measures (Bugs, Difficulty, Effort, Length, Time, Vocabulary, Volume, information about distinct identifiers such as Operands and Operators), and number of lines of code (LOC). Maintainability is also evaluated for some code structures, such as classes.

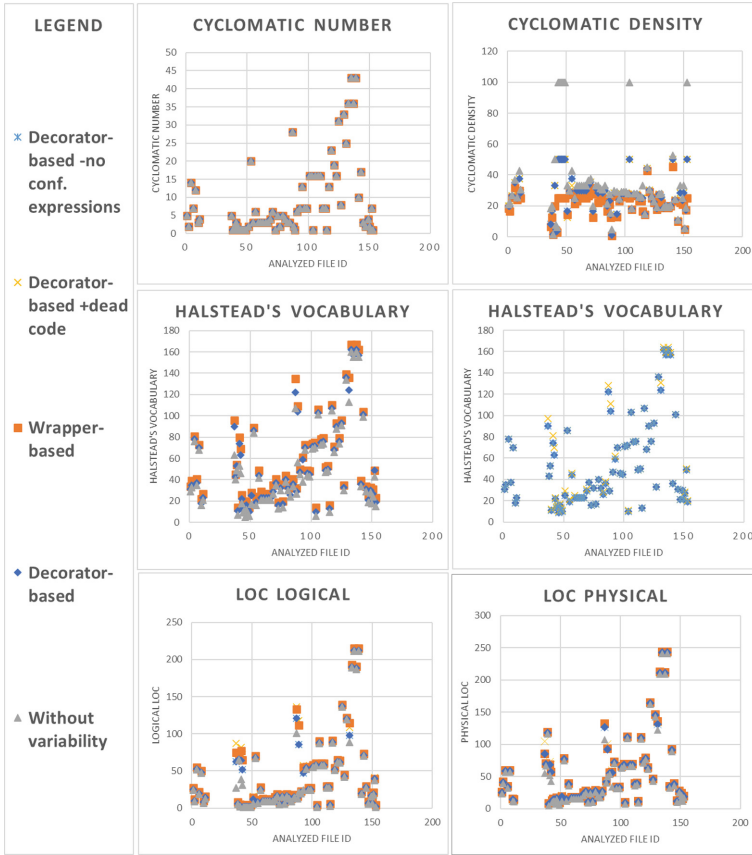


Fig. 4. Pair comparison of evaluated particular complexity metric on variability annotated files among all introduced cases: Part 2

As has been mentioned in Sect. 3, the study was performed on a software product line aimed at graphical applications, which we developed for evaluation purposes.

We iterated with the design of annotations (detachable decorators) with available decorators in TypeScript to cover all variable parts more concisely. One-line declarations, calls, imports, and other code fragments showed to be problematic to decorate. Most of these code fragments cannot be wrapped inside functions. Additionally, it's not feasible to study complexity due to unavailable tools supporting illegal decorators in TypeScript. Despite this, we used illegal experimental decorators that annotated newly declared variables and put them under one-line variable code fragment that needed to be covered. In the next step, they are transformed into a wrappers that can be compiled, especially for the evaluation phase. For some cases, such as in Angular, the compiler can ignore

the illegal use of decorator with `@ts-ignore`, thus allowing native development of applications. The code is still slightly polluted with necessary dead code.

For each target project file, we created a hierarchic structure from metrics being assessed with the possibility of applying various operations, such as a difference between two such structures. Finally, results are divided into files, classes, their methods, and averages when many classes or methods are inside of given files. Due to the domain orientation/solution design, the results strongly depend on the given variability markings and the technology and frameworks used. A different domain or solution design can require a different number of annotations that can vary in their various types. For this purpose, the analysis is applied to files of software product line rather than on their merged version and then statistically evaluated. In addition, the resulting complexities are divided by the number of decorators used. Consequently, the number and complexity of evaluated code constructs should not significantly differ from other associated and similarly analyzed files after transformation is applied into a different form. Still, the more use cases are implemented, the more complex variability expressions and more dense annotations are, especially for contradicting features. Accordingly, the resulting complexities are also affected. These initial conditions do not prevent observing variability management's additive effect on code complexity. Additionally, various code-based variability annotations are compared based on their complexities and evaluated to see if unsupported illegal decorators or their future versions will improve complexity and quality.

Different complexity metrics extracted from the code transformed according to proposed cases are quantitative, except a list of used identifiers and operators in the Halstead measures. Additionally, performed normality tests do not confirm normality; thus, a test independent of probability distribution is chosen. After getting information about possible correlations, all proposed metrics were shown to be dependent. It also holds for measurements that compare the code in a particular case (sample) and its transformed/extended version (opposite sample). Thus, pair testing is used. Whether compared complexity measures significantly differ after and before applying the particular case of variability management annotations helps to find unaffected measures or those with marginal values and is verified with the paired nonparametric Wilcoxon test. If the significance level or p-value is below the given threshold of 0.05, the zero hypothesis is rejected, and the samples are significantly different. Another option is to use a Kruskal-Wallis test, which uses the Nemenyi method to determine if a change is also significant.

In this experiment, over 76 variability annotated files that contain at least one variability annotation are tested. Additionally, the 84 classes stored in the mentioned files were used to test their in-class complexities, but only 64 were unique. The differences between each file/member pair cases (in each vertical line of the graph) are visualized in Figs. 3 and 4.

4.1 Code Complexity of Variability Configuration Expressions

Our primary focus was to evaluate whether changes to different complexity measures are significant after inserting variability configuration expressions. Quantitative results also help optionally optimize used constructs or recommend their use. The solution with configuration expressions seems more complex than the version without them. Halstead measures, such as Bugs and Length, always increased after introducing configuration expressions. Halstead's Vocabulary and Identifiers, such as operands and operators, always increased or remained unchanged. They remained unchanged due to using empty JSON in cases where the whole file should be copied according to our variability management mechanism policy. The logical LOC has a similar tendency. The cyclomatic complexity and the number of physical LOC remained unchanged. Halstead's Difficulty, Effort, and Time are decreased or increased for some measures.

Table 1. Code complexity for Case 1 and 2 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p>0.05
Cyclomatic Complexity	1.0000	0	1.0000E+00	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.9897	0	2.1335E-04	-2.12, -0.7690	-1.4110	FALSE
Halstead's Bugs	0.9989	2926	1.9956E-14	0.002, 0.005	0.002	FALSE
Halstead's Difficulty	0.9976	294	3.8958E-02	0.07, 0.59	0.3265	FALSE
Halstead's Effort	0.9992	2652	7.5836E-10	57, 103	79.7495	FALSE
Halstead's Length	0.9988	2926	1.2543E-15	1.00, 2.50	1.0001	FALSE
Halstead's Time	0.9992	2652	7.5836E-10	3.19, 5.74	4.4308	FALSE
Halstead's Vocabulary	0.9990	435	1.9687E-06	1.50, 3.50	2.0000	FALSE
Halstead's Volume	0.9903	2926	3.6708E-14	35.18, 45.23	38.6940	FALSE
Halstead's Id Dist. Operands	0.9984	171	1.3795E-04	2.00, 4.00	3.5001	FALSE
Halstead's Id Ttl Operators	0.9981	171	1.5794E-04	2.00, 13.50	9.0000	FALSE
Halstead's Id Dist. Operators	0.9993	66	1.0893E-03	NaN, NaN	1.0000	FALSE
Halstead's Id Ttl Operators	0.9982	2926	1.2460E-15	1.00, 1.50	1.0001	FALSE
LOC Physical	1.0000	0	NaN	NaN, NaN	NaN	TRUE
LOC Logical	0.9978	171	1.5853E-04	1.00, 7.00	5.0000	FALSE

We tested the significance of these changes in the paired test on a significance level of 0.05. The results are shown in Table 1. Only the cyclomatic complexity and number of logical LOC do not deny Hypothesis 0, thus confirming that these measures are not significantly different except the remaining ones. Accordingly, our Hypothesis 1 is rejected. Evaluating complexities from classes showed that Halstead's Difficulty is near the boundary of being significantly different and can be improved by removing redundant classes used only to annotate and preserve given files during variability handling. Class maintainability is also evaluated and remains without significant difference in this case.

4.2 Wrappers Vs. Detachable Decorators

The similarly applied paired Wilcoxon test on Case 3 and 1 pair proved that the wrappers and detachable decorators significantly differ in all complexity metrics except the cyclomatic complexity. The remaining code fragments are more complex for wrappers by reaching higher values for some Halstead complexity measures (Bugs, Length, Operator and Operand identifiers, Volume, and Vocabulary, except one sample). LOC (physical and logical) followed this tendency despite a few similar values. Not always, but complexity is high for other Halstead measures (Difficulty, Effort, and Time). Declaring new variables is affected by the complexity of the whole file and results in different complexities. Halstead's Bugs, Length, and Volume are mainly increased when their content is wrapped inside a method and then when classes are wrapped. These preferences are for Vocabulary usually swapped. The simpler the wrapped code is, the lower the differences are. Only cyclomatic density is decreased.

Table 2. Code complexity for Case 3 and 1 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p > 0.05
Cyclomatic Complexity	1.0000	0	1.0000E+00	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.8226	0	3.5776E-13	-4.1959, -2.28	-3.02	FALSE
Halstead's Bugs	0.9997	2556	2.4526E-13	0.01, 0.02	0.0141	FALSE
Halstead's Difficulty	0.9971	2237	5.9298E-09	0.60, 0.80	0.7390	FALSE
Halstead's Effort	0.9988	2386	2.2106E-10	503.04, 1493.10	841.6983	FALSE
Halstead's Length	0.9997	2556	9.3382E-17	6.00, 6.00	6.0000	FALSE
Halstead's Time	0.9988	2386	2.2106E-10	27.95, 82.95	46.7609	FALSE
Halstead's Vocabulary	0.9994	2484	4.2116E-14	3.00, 3.45	3.0000	FALSE
Halstead's Volume	0.9997	2556	2.4761E-13	39.32, 45.96	42.2363	FALSE
Halstead's Id Dist. Operands	0.9996	2415	2.5713E-16	2.00, 2.00	2.0000	FALSE
Halstead's Id Ttl Operands	0.9999	2556	9.3382E-17	2.00, 2.00	2.0000	FALSE
Halstead's Id Dist. Operators	0.9886	2030	2.1410E-12	1.00, 1.50	1.0000	FALSE
Halstead's Id Ttl Operators	0.9999	2485	9.8502E-17	4.00, 4.00	4.0000	FALSE
LOC Physical	0.9998	2556	2.1563E-16	1.00, 1.00	1.0000	FALSE
LOC Logical	0.9999	2485	9.8502E-17	2.00, 2.00	2.0000	FALSE

The Wilcoxon test confirmed the significance of these differences in all cases except the cyclomatic complexity. In conclusion, Hypothesis 2 cannot be rejected. Consequentially, their significance level is too far from the threshold of 0.05, which makes the detachable decorators (Case 1) preferable over the wrappers (Case 3). Additionally, the full support of decorators for one-line constructs will probably improve results when they are used during comparison with wrappers and deepen the differences between both versions. The cyclomatic complexity is not affected by any change inside a given file caused by the variability management's independence. Results are displayed in Table 2.

4.3 Significance of Variability Management Code Complexity

Hypothesis 3 is proposed to test the significance of the complexity of various variability constructs from Case 1 to each overall business code complexity (Case 4) for each file. The improvements are presented in Table 3 and showed that detachable decorators strongly affect the majority of complexity measures except for cyclomatic complexity (no change) and Halstead’s Difficulty. Some complexity metrics contribute in both directions, especially some Halstead measures (Difficulty, Effort, and Time) measured for each analyzed file. Cyclomatic density is the only one that decreased after the variable code fragments were removed.

Table 3. Code complexity for Case 1 and 4 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p > 0.05
Cyclomatic Complexity	1.0000	0	NaN	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.9264	0	3.6200E-14	−4.63, −2.05)	−2.9825	FALSE
Halstead’s Bugs	0.9900	2926	3.6200E-14	0.01, 0.02	0.0130	FALSE
Halstead’s Difficulty	0.9921	1489	8.9495E-01	−0.19, 0.52	0.0280	TRUE
Halstead’s Effort	0.9920	2486	1.2000E-07	123.31, 199.05	155.8794	FALSE
Halstead’s Length	0.9885	2926	1.2500E-15	5.00, 6.50	5.0001	FALSE
Halstead’s Time	0.9920	2486	1.2000E-07	6.85, 11.06	8.6572	FALSE
Halstead’s Vocabulary	0.9880	2926	1.2900E-14	3.00, 3.50	3.0000	FALSE
Halstead’s Volume	0.9903	2926	3.6700E-14	35.18, 45.23	38.6939	FALSE
Halstead’s Id Dist. Operands	0.9855	2926	1.2500E-15	2.00, 3.00	2.0001	FALSE
Halstead’s Id Ttl Operands	0.9891	2926	1.2500E-15	2.00, 3.00	2.0001	FALSE
Halstead’s Id Dist. Operators	0.9928	406	2.6600E-06	1.50, 2.00	1.9999	FALSE
Halstead’s Id Ttl Operators	0.9863	2926	1.2500E-15	3.00, 3.50	3.0001	FALSE
LOC Physical	0.9904	2926	1.0300E-16	2.00, 2.00	2.0000	FALSE
LOC Logical	0.9734	2926	1.2500E-15	1.00, 1.50	1.0001	FALSE

The wrapped code significantly affects (from half to more than twice for the one annotated code fragment, but more than twice for the standalone file) Halstead’s Bugs, Length, Vocabulary, Operator and Operand identifiers, and Volume. Both the logical and physical number of LOCs are minimally doubled. In the case of Halstead’s Bugs, the lowest contributions are associated with statements only used to support variability, as mentioned before. Consequently, using supporting code for variability management can be limited, and illegal detachable decorators seem preferable to wrappers. The most affected are modules that should be configured differently to reduce additional variables, “if”, “than”, “else” constructs, and necessarily associated import statements. The last one is the most important to handle because of the necessity to use wrappers; otherwise, they must be held at least with illegal detachable decorators. Their nature is thus not intended directly to handle variability.

Additionally, the removal of files with the most wrappers decreases most Halstead measures. The test on significance between the state before and after applying variability management was performed again, and results are displayed in Table 4. The significance level improved in most cases by about a hundredfold (expressed in bold italics) and tenfold (bold) for the remaining ones, except for cyclomatic complexity and physical LOC. This test showed how reducing wrapper parts in cases such as module and routing configuration or services with additional code handling in our software product line aimed at graphical applications can help reduce the code complexity of used variability management constructs to mark variable code. Still, the change is too far from the threshold of 0.05. In summary, using variability management constructs significantly affects most used complexity measures and cannot decrease them by any available code construct below the significance threshold. Hypothesis 3 is not rejected, thanks to Halstead’s Difficulty. Still, less complex variability-aware code with higher quality can be produced in this restricted way.

4.4 The Effect of Dead Code Constructs on Code Complexity

The next step is to discover how redundant dead code affects the analyzed complexity measures in order to validate Hypothesis 4. This code is used in helper functionality based on illegal decorators in TypeScript that mediate visually positioning variability annotations next to the variable code fragments as an alternative to wrapper constructs. The solution required using these constructs in only 9 out of all 76 files. Subtracting each of the complexity measures of Case 5

Table 4. Code complexity for Case 1 and 4 compared without most of the files with wrappers.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p > 0.05
Cyclomatic Complexity	1.0000	0	NaN	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.9277	0	<i>1.6427E-12</i>	-4.58, -1.81	-2.7695	FALSE
Halstead’s Bugs	0.9969	2211	<i>1.6442E-12</i>	0.01, 0.01	0.0120	FALSE
Halstead’s Difficulty	0.9948	886	1.6171E-01	-0.28, 0.11	-0.1545	TRUE
Halstead’s Effort	0.9979	1787	<i>1.3588E-05</i>	102.09, 152.93	126.98	FALSE
Halstead’s Length	0.9965	2211	2.5043E-14	5.00, 5.00	5.0000	FALSE
Halstead’s Time	0.9979	1787	<i>1.3588E-05</i>	5.67, 8.50	7.0545	FALSE
Halstead’s Vocabulary	0.9963	2211	4.1183E-13	2.50, 3.50	2.9999	FALSE
Halstead’s Volume	0.9969	2211	<i>1.6743E-12</i>	32.77, 38.41	35.4739	FALSE
Halstead’s Id. Dist. Operands	0.9953	2211	2.5043E-14	2.00, 2.00	2.0001	FALSE
Halstead’s Id. Ttl Operands	0.9954	2211	2.5043E-14	2.00, 2.00	2.0001	FALSE
Halstead’s Id. Dist. Operators	0.9958	171	<i>1.4868E-04</i>	2.00, 3.00	2.0000	FALSE
Halstead’s Id. Ttl Operators	0.9953	2211	2.5043E-14	3.00, 3.00	3.0001	FALSE
LOC Physical	0.9980	2211	7.4931E-16	2.00, 2.00	2.0000	FALSE
LOC Logical	0.9958	2211	2.5043E-14	1.00, 1.00	1.0001	FALSE

with unwanted helper code from the case without them measures their overhead. Complexity is again increased in most Halstead metrics (Bugs, Effort, Length, Time, Vocabulary, and Volume) and LOC metrics (logical and physical). Only cyclomatic density decreased in all cases. The results of the paired Wilcoxon test that tests the significance of the change for most of the complexity metrics are shown in Table 5. Hypothesis 4 cannot be rejected. Still, compared with previous measurements, the significance level is not far from the threshold of 0.05 for most evaluated complexity metrics.

Table 5. Code complexity for Case 5 and 1 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p > 0.05
Cyclomatic Complexity	1.0000	0	1.0000	NaN, NaN	NaN	TRUE
Cyclomatic Density	1.0000	0	0.0092	-1.11, -0.25	-0.495	FALSE
Halstead's Bugs	0.9998	45	0.0092	0.01, 0.04	0.0205	FALSE
Halstead's Difficulty	0.9999	40	0.0440	0.01, 0.53	0.2210	FALSE
Halstead's Effort	0.9996	45	0.0092	551.81, 2876.1	1199	FALSE
Halstead's Length	0.9998	45	0.0091	4.00, 16.00	9.0001	FALSE
Halstead's Time	0.9996	45	0.0092	30.66, 159.78	66.603	FALSE
Halstead's Vocabulary	0.9999	45	0.0034	NaN, NaN	1.0000	FALSE
Halstead's Volume	0.9998	45	0.0092	32.04, 108.08	62.861	FALSE
Halstead's Id Dist. Operands	0.9999	45	0.0034	NaN, NaN	1.0000	FALSE
Halstead's Id Ttl Operands	0.9996	45	0.0091	2.00, 8.00	4.5000	FALSE
Halstead's Id Dist. Operator	1.0000	0	1.0000	NaN, NaN	NaN	TRUE
Halstead's Id Ttl Operators	0.9999	45	0.0091	2.00, 8.00	4.5000	FALSE
LOC Physical	0.9990	45	0.0092	2.50, 12.00	5.5000	FALSE
LOC Logical	0.9996	45	0.0091	2.00, 8.00	4.5000	FALSE

5 Discussion

The experiments depend on variability management policies, which prescribe how information about variability is marked. Our software product line aimed at graphical applications is based on our method of developing lightweight aspect-oriented software product lines with automated product derivation and its policies [21].

Additionally, the values of specific metrics can differ under different variability configurations according to the number and complexity of variable features. Only five features and five subfeatures were implemented, along with one annotation per file for the initial configuration of variability management, which proved significantly different from the original business code. Modern software product

lines count from tens to thousands of features [13], where many are scattered across multiple files and commits during evolution [10]. Consequently, this introduces numerous variability constructs with more complex configuration expressions. Managed modules in the Angular framework require annotating various fragments. On the contrary, after removing such cases, the result of the presented test is still closely behind the threshold to reject a zero hypothesis and proves a significant difference. Still, the complexities can be improved even with minor changes. Some Halstead metrics can lead to unclear properties for small or middle programs, possibly affecting their interpretation [29]. Consequently, the measured values should be verified with user testing, especially for variability management. Another option is to use them according to updated versions on a large scale.

Additional metrics should be introduced to fully observe the comprehension of particular variability management constructs, especially expressions by developers. Variability expressions are usually mapped to features positioned in hierarchical feature models or belonging to the chains of dependencies amongst features. These relations are not considered and pose the problem of constructional validity.

The detachable decorators even empirically overcomes the wrappers when the ending part should be necessarily included, the nesting of wrappers is error-prone and hard to read, and even they cannot be directly bound to a particular entity in code. All LOC and Halstead metrics agreed for conclusion validity, but complications emerged with cyclomatic complexity. Specifically, conditional reasoning is affected by variability conditions, but they do not affect cyclomatic complexity. This measure failed in all introduced cases due to a misunderstanding of the role of variability management constructs.

We observed that the complexity of detachable decorators as constructs bound to other code constructs depends on the complexity of these constructs. Consequently, we found the problem of internal validity because our constructs are easily detachable without any side effects and thus not bound to business logic with its flows. On the contrary, the more modular the annotated business entity is, the better these constructs are for modularization. Specifically, code-complexity metrics are unaware of variability management and pose the problem of inner validity. New metrics should be introduced for this purpose.

The problem of generalizing outcomes for constructs belonging to other programming languages or computation models under specific conditions falls into external validity. For example, transitions of finite automaton or state machine can be aggregated or grouped. Still, it is useless to annotate them individually if simulators in this process do not support advanced data structures. Unknown relations to variability or immature technologies cause these problems.

6 Related Work

The most similar expressions to configuration expressions are presence conditions [4]. They manage the presence of particular functionality in various models according to the configuration of features [4], usually through feature models.

The same operators (and/or) are used with configuration in variability management on various models. Their known drawback is the ability to manage the presence only of available functionality, negative variability [24]. Compared with our JSON [21] configuration expressions instantiated as objects in JavaScript/TypeScript, a presence condition cannot easily express hierarchic information and even hold more information, such as additional strings or values assigned during development. If applied at the code level, the variables used in these presence conditions usually interleave with the code [21]. To the best of our knowledge, their code complexity was never assessed nor used to optimize them.

Many authors focused on short programs [12, 19, 20] possibly because of the lack of an automated support for their studies. However, we developed such support, which made possible for our study to be of a nontrivial size.

Athar Khan et al. compare three programming languages according to measured metrics on the implementation of a sorting algorithm [12]. LOC, cyclomatic complexity, and the Halstead measures were used to determine the most complex implementation and implications for testing time. Exceptionally for the Halstead difficulty, the different most complex candidate programming language was determined [12].

Peitek et al. observed the relation between metrics and brain activity [20]. The fMRI study [20] found only a small correlation of the LOC and Halstead measures with the response time and a medium correlation with the correctness of the resulting program outputs. The medium correlation also held for brain activation and deactivation, while cyclomatic complexity is not correlated in any case. The reason for using the Halstead measures in this study is given by the increase in cognitive processing demands due to the number of symbols [25]. Similarly, cyclomatic complexity was used to analyze control flows due to their effects on rule-guided conditional reasoning [14, 16]. Our study could be extended to evaluate the level of cognitive processing and conditional reasoning in different approaches to variability management.

Sehgal and Mehrotra [26] used Halstead's volume to predict faults among system components in the reliability and mean time between failures metric, which considers usage time and the amount of code with operators and operands. Graves et al. [8] showed how complexity metrics correlate with each other in a similar way our study indicated.

7 Conclusions and Future Work

This paper presents a study on how selected approaches to expressing variability in code affect code complexity. To evaluate and compare the complexity of essential aspects of different approaches to in-code variability management, we designed five prominent cases of how variability is expressed in code: using decorators, using decorators without variability configuration expressions, using wrappers, using decorators with additional unwanted dead code constructs not being included for illegal decorators, and with no variability expressed in code at all. To measure code complexity in these cases, we used a framework for evaluating TypeScript code that we implemented in Java. Our framework is capable

of assessing 15 metrics, comprising several variants of the LOC, Halstead, cyclomatic complexity, and cyclomatic density metrics. Decorators are detachable because they are decorating a particular code construct with a predefined naming convention and have no effects on code. The study was conducted on a software product line aimed at graphical applications we developed for evaluation purposes.

We came to a range of interesting findings. Detachable decorators proved to be significantly less complex than other ways of expressing variability in code, especially than the in-code version of wrappers. Except for dynamic self-adaptability, annotations in comments (as in `pure::variants`) do not directly affect the complexity of business functionality. Similarly, decorators can be entirely separated from business logic and their names are easily adaptable to particular domains or extensions. They are even applicable to incorporating changes dynamically on the fly and their application is forced to improve modularity. Configuration expressions are put inside their arguments more concisely in the JSON format. On the contrary, illegal decorators are still necessary for configuration purposes or exceptional cases, rather than traditional wrappers to annotate import statements. Additionally, introducing redundant code for these purposes significantly affects complexity measures. However, even decorators proved significantly more complex for various metrics than not having variability expressed in code at all.

The configuration expressions used for their hierarchic nature to concisely express feature models significantly increase most code complexity measures, especially the Halstead ones. Consequently, optimization to more comprehensive versions and even to make easier creating autonomous processes of optimizing configuration expressions and evolving software product lines are required to configure, simulate, and evaluate effects on the extensive number of features. In our future work, the results can help design a tool to visualize only certain views of variability on the fly and update particular configuration expressions accordingly. Additionally, we want to automatically optimize configuration expressions and evaluate which hierarchically expressed configuration made according to feature model fits the best.

Acknowledgements. The first author was supported by the STU Grant Scheme for Support of Young Researchers.

References

1. Blair, L., Pang, J.: Aspect-oriented solutions to feature interaction concerns using AspectJ, p. 17 (2003)
2. Cardoso, J.: How to use decorators in TypeScript. digitalocean.com (2021). <https://www.digitalocean.com/community/tutorials/how-to-use-decorators-in-typescript>
3. Chen, L., Ali Babar, M., Ali, N.: Variability management in software product lines: a systematic review. In: Proceedings of the 13th International Software Product Line Conference, pp. 81–90 (2009)

4. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005). https://doi.org/10.1007/11561347_28
5. Fenske, W., Thüm, T., Saake, G.: A taxonomy of software product line reengineering. pp. 1–8. ACM, Sophia Antipolis France (2014)
6. Figueiredo, E., Cacho, N., Sant’Anna, C., Monteiro, M.: Kulesza: evolving software product lines with aspects: an empirical study on design stability. In: Proceedings of 30th International Conference on Software Engineering, ICSE’08. ACM (2008)
7. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness (2001)
8. Graves, T., Karr, A., Marron, J., Siy, H.: Predicting fault incidence using software change history. *IEEE Trans. Software Eng.* **26**(7), 653–661 (2000)
9. Heidenreich, F., Kopcssek, J., Wende, C.: FeatureMapper: mapping features to models. In: In: Proceedings of 30th International Conference on Software Engineering. ICSE 2008, Leipzig, Germany, pp. 943–944 (2008)
10. Hinterreiter, D., Grünbacher, P., Prähofer, H.: Visualizing feature-level evolution in product lines: a research preview. In: Madhavji, N., Pasquale, L., Ferrari, A., Gnesi, S. (eds.) REFSQ 2020. LNCS, vol. 12045, pp. 300–306. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44429-7_21
11. Kasto, N., Whalley, J.: Measuring the difficulty of code comprehension tasks using software metrics **136** (2013)
12. Khan, A.A., Mahmood, A., Amralla, S.M., Mirza, T.H.: Comparison of software complexity metrics. *Int. J. Comput.* **04**, 19–26 (2016)
13. Khan, F., Musa, S., Tsaramirsis, G., Buhari, S.: SPL features quantification and selection based on multiple multi-level objectives. *Appl. Sci.* **9**, 18 (2019)
14. Kulakova, E., Aichhorn, M., Schurz, M., Kronbichler, M., Perner, J.: Processing counterfactual and hypothetical conditionals: an fMRI investigation. *Neuroimage* **72**, 265–271 (2013)
15. Leahy, M.: TyphonJS-ESComplex (2018). <https://www.npmjs.com/package/typhonjs-escomplex>
16. Liu, J., Zhang, M., Jou, J., Wu, X., Li, W., Qiu, J.: Neural bases of falsification in conditional proposition testing: evidence from an fMRI study. *Int. J. Psychophysiol.* **85**(2), 249–256 (2012)
17. Loughran, N., Rashid, A.: Framed aspects: supporting variability and configurability for AOP. In: Bosch, J., Krueger, C. (eds.) ICSR 2004. LNCS, vol. 3107, pp. 127–140. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27799-6_11
18. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting product line evolution with framed aspects, p. 5 (2004)
19. Muriana, B., Paul Onuh, O.: Comparison of software complexity of search algorithm using code based complexity metrics. *Int. J. Eng. Appl. Sci. Technol.* **6**(5), 24–29 (2021)
20. Peitek, N., Apel, S., Parnin, C., Brechmann, A., Siegmund, J.: Program comprehension and code complexity metrics: An fMRI study. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE 2021, Madrid, ES, pp. 524–536. IEEE (2021)
21. Perdek, J., Vranić, V.: Lightweight aspect-oriented software product lines with automated product derivation. In: Abelló, A., et al. (eds.) ADBIS 2023. CCIS, vol. 1850, pp. 499–510. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-42941-5_43

22. Perdek, J., Vranić, V.: Matrix based approach for structural and semantic analysis supporting software product line evolution. In: Proceedings of the Tenth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2023. CEUR Workshop Proceedings, Bratislava (2023)
23. pure::systems: PLE & code-managing variability in source code (2020). <https://youtu.be/RIUYjWhJFkM>
24. Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (2011)
25. Schuster, S., Hawelka, S., Himmelstoss, N.A., Richlan, F., Hutzler, F.: The neural correlates of word position and lexical predictability during sentence reading: evidence from fixation-related fMRI. *Lang. Cogn. Neurosci.* **35**(5), 613–624 (2020)
26. Sehgal, R., Mehrotra, D.: Predicting faults before testing phase using Halstead's metrics. *Int. J. Software Eng. Appl.* **9**, 135–142 (2015)
27. Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A classification of product sampling for software product lines. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, New York, NY, USA, pp. 1–13. SPLC '18, Association for Computing Machinery (2018)
28. Woods, J.: Possible ES6 extensions (2021), <https://github.com/tc39/proposal-decorators/blob/master/EXTENSIONS.md>
29. Zuse, H.: Resolving the mysteries of the halstead measures (2005)