Slovak University of Technology in Bratislava

Faculty of Informatics and Information Technologies

Ing. Jakub Perdek

Dissertation Thesis Abstract

# Aspect Oriented Knowledge-Driven Evolution of Software Product Lines With Hierarchically-Expressed Variability Information Preserved in Code

to obtain the academic title of *Philosophiae doctor* (PhD.)

| | |
|---|---|
| Study Program: | Applied Informatics |
| Field of Study: | Computer Science |
| Place of development: | Institute of Informatics, Information Systems and Software Engineering<br>Faculty of Informatics and Information Technologies<br>Slovak University of Technology in Bratislava |

**Bratislava, July 2025**

Dissertation Thesis has been developed at the Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava.

| | |
|---|---|
| **Submitter**: | Ing. Jakub Perdek<br>Institute of Informatics, Information Systems<br>and Software Engineering<br>Faculty of Informatics and Information Technologies<br>Slovak University of Technology in Bratislava |
| **Supervisor**: | Doc. Ing. Ján Lang, PhD.<br>Institute of Informatics, Information Systems<br>and Software Engineering<br>Faculty of Informatics and Information Technologies<br>Slovak University of Technology in Bratislava |
| **Former Supervisor and Consultant**: | Prof. Ing. Valentino Vranić, PhD.<br>Faculty of Informatics<br>Pan-European University, Bratislava, Slovakia |
| **Oponents**: | doc. Dr. Miloš Dobrojević<br>Faculty of Informatics and Computing<br>Sinergija University, Republika Srbská, Bosna a Hercegovina<br><br>doc. Ing. Csaba Szabó, PhD.<br>Faculty of Electrical Engineering and Informatics<br>University of Technology in Košice, Slovakia |

Dissertation Thesis Abstract was sent: ....................
Dissertation Thesis Defence will be held on ......... at ........ pm at the Institute of Computer Engineering and Applied Informatics, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava (Ilkovicova 2, Bratislava).

**Prof. Ing. Ivan Kotuliak, PhD.**
Dean of FIIT STU in Bratislava

# ANNOTATION

Slovak University of Technology Bratislava
Faculty of Informatics and Information Technologies

| | |
|---|---|
| Author: | Ing. Jakub Perdek |
| Supervisor: | Doc. Ing. Ján Lang, PhD. |
| Former Supervisor and Consultant: | Prof. Ing. Valentino Vranić, PhD. |
| Degree Course: | Applied informatics |
| Title: | Aspect Oriented Knowledge-Driven Evolution of Software Product Lines With Hierarchically-Expressed Variability Information Preserved in Code |
| July 2025 | |

Variability handling and reuse are methodologically and practically managed in software product lines. In accordance with their taxonomy, the annotation-based type preserves the annotations of variability in code while ensuring the establishment of highly configurable systems out of its features. Despite the benefits of annotation-based software product lines, in-code variability management is not adjusted to concisely preserve hierarchical information in the code in a lightweight, minimalistic, and fully automated manner. Specifically, annotations are left without a selection procedure based on in-code complexity, policy enforcement, naming conventions, managing domain knowledge, and criteria for maintaining feature models directly in the code. Consequently, we introduce a methodology for the establishment of a software product line in a lightweight fashion, a framework with competing strategies based on in-code complexities concerning the context of the entire code fragment to select the annotation with the lowest complexity, a scalable solution infrastructure by fast matrix-based methods for graph matching and clustering, and finally most of them entirely integrated into fully automated and minimalistic variability management. Additionally, the methodology is verified on established software product lines prepared in accordance with the variability managed in the code, but primarily on the minimalistic and fully automated version for managing fractals driven by structural information towards its semantic extensions. Through minimalistic fully automated evolution, we achieved the ability to handle variability from the early beginning while reaching reuse, massive production, and discovering domain knowledge. Our methodology provides new perspectives to solve existing challenges with diverse artifacts, including the simulation of feature interactions owing to automated scenario generation from software product line evolution or decision-making with a custom-designed model while incorporating new features. A notable contribution of our work lies in elaborating the methodology of preserving feature models in code, with its verification in automated minimalistic evolution.

# ANOTÁCIA

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

| | |
|---|---|
| Autor: | Ing. Jakub Perdek |
| Školiteľ: | Doc. Ing. Ján Lang, PhD. |
| Predchádzajúci Školiteľ a Konzultant: | Prof. Ing. Valentino Vranić, PhD. |
| Študijný program: | Aplikovaná Informatika |
| Názov Práce: | Evolúcia Radov Softvérových Výrobkov s Variabilitou v Kóde Nízkotonážnym a Automatizovaným Spôsobom |
| Júl 2025 | |

Správa variability spolu so znovupoužitím je metodologicky a prakticky riadená v rámci existujúcich radov softvérových výrobkov. Jeden zo spôsobov realizácie podľa taxonómie predstavuje použitie anotácií pri zavádzaní vysoko konfigurovateľných systémov z dostupných vlastností. Napriek známym výhodám, manažovanie variability v kóde nie je prispôsobené pre prehľadné a zrozumiteľné udržovanie hierarchickej informácie v kóde. To môže byť naviac dosiahnuté jednoduchým, minimalistickým, a plne automatizovaným spôsobom. Napríklad existujúce prístupy neumožňujú porovnať rôzne druhy anotácií a výrazov na základe zložitosti v kóde, vynútiť pravidlá v kóde, použiť menné politiky, spravovať doménové znalosti, a nie sú identifikované kritériá pre flexibilné udržiavanie modelov vlastností v kóde. Predstavujeme preto metodológiu umožňujúcu zavedenie radu softvérových výrobkov odľahčeným, nenáročným spôsobom, rámec implementujúci stratégie pre určenie druhu anotácie pre správu variability s najnižšou zložitosťou určenou zo zdrojového kódu a jeho kontextu, škálovateľnú infraštruktúru zabezpečenú integráciou rýchlych grafovo-orientovaných metód pre zhodu a hierarchické zhlukovanie uzlov, a kompletnú integráciu vymenovaných metodológií do plne automatizovanej a minimalistickej správy variability. Predstavená metodológia je overená na navrhnutých a zhotovených radoch softvérových výrobkov s cieľom spravovať variabilitu v kóde, primárne na minimalistickej a plne automatizovanej verzii aplikovanej na evolúciu fraktálov pri ktorej každé rozšírené fraktálu je riadené podľa štruktúrnych indikátorov a rozšíriteľné smerom ku sémantickým. Prostredníctvom minimalistickej plne automatizovanej evolúcie sme už od skorých počiatkov vývoja zabezpečili správu variability za súčasného znovupoužitia, masívnej produkcie, a objavovania doménových znalostí (pri iteratívnej integrácii nových vlastností). Naša metodológia poskytuje perspektívy pre riešenie existujúcich výziev použitím rôznorodých artefaktov, napríklad pri simulácii interakcií vlastností vďaka automatizovanému generovaniu scenárov získaných z evolúcie radov softvérových výrobkov alebo rozhodovania pri zapracovaní nových vlastností vďaka navrhnutiu špecifického rozhodovacieho modelu. Perspektívny prínos našej práce pozostáva z vypracovania metodológie pre udržiavanie modelov vlastností v zdrojovom kóde spolu s verifikáciou jeho spôsobilostí v rámci plne automatickej minimalistickej evolúcie.

# Table of Contents

# 1. Introduction

Software product lines increase the effectiveness of product delivery by reusing common assets, including software artifacts, architectural decisions, written tests, and others. The principle of their success lies in the reuse efficiency, which is achieved by not repeating the analysis stage for developing each product but applying domain knowledge instead. Reuse is ensured in both phases—domain and application engineering [12]. Generalization and decomposition are applied in the first phase to create generators and reusable components while covering as much variation as possible. The focus is on applying domain knowledge to separate what is common from what is variable. The second phase transforms generated artifacts and developed components from the first phase into final products and continues with their customization [12].

Variability handling is the primary concern in software product lines based on determining what is common and variable in terms of features. Features are in literature defined as user-visible aspects, qualities, or characteristics of a software system [11], and as logical units of behavior given by the set of requirements elsewhere [5], are identified and realized later in the first domain engineering phase. Common and varying requirements as representatives of problem space are transformed into reusable components in solution space in a process called derivation (of a software product) [13].

Reliance on expert knowledge, primarily associated with variability handling in software product lines, is crucial and leads to the emergence of various challenges. The majority of solutions neglect knowledge modeling and simulation of interactions between features [9, 20]. Opportunities to handle defects and provide quality assurance are missed or do not rely on the knowledge within the software family. Similarly, documented are problems with comprehending interactions between variants [37] caused by the rapid increase of mutual relations and dependencies with new features. Similarly, variability handling is not fully covered in general primarily because it does not get beyond software product lines [20].

## 1.1 Thesis Statement

In annotation-based software product lines, variability is directly expressed in code using annotations. Such a code is difficult to understand, neglecting the benefits of having variability directly managed in code. A particular problem is that variability handling constructs are mixed with business logic. The effects of how we comprehend variability in code are unrecognized, leaving no opportunity for additional improvements, even in the case of implications for automated evolution. Constraints and variable configurations must be known to incorporate this change manually. When the enumeration of constraints or incorporation of domain knowledge is directly preserved in the code, automated updates are simplified from one place. Specifically, no additional language is necessary to select and quantify the affected locations in the code, as it is in aspect-oriented programming. Preserving additional information has the ability to enforce policies in a particular place in the code, but it also increases complexity and loses the benefit of preserving such information independently. Furthermore, current approaches to developing annotation-based software product lines do not support the expression of the hierarchical structure of variability, which is its inherent property.

Additionally, annotations such as tags in frame technology can still be affected from outside using aspect-oriented programming. Consequently, the simplicity of the managed variability can then be enriched with highly complex constructs affecting code while benefiting from reuse. Reuse is achieved primarily by non-invasive integration of concerns and separation of composition rules as part of handled variability. Reusing existing functionality is a step to be applied to oneself in a recursive manner or to handle specific cases.

To a large extent, features in product lines come from the domain knowledge. Extensive software product lines involve a large number of features with complex variability relations between them. Handling this manually is difficult and error-prone in general, and even more so if this has to happen directly in code as in annotation-based software product lines. Subsequently after handling variability on low code level in parallel with extracting appropriate knowledge from this step, the aspect-oriented nature supported with this knowledge would less incline to break modularization of resulting solution.

Taking all this into account, the following thesis can be stated:

> Evolving software product lines would benefit from in-code separation of variability from business logic, preserving the hierarchical structure of variability in the source code, and automating the derivation of features from the domain knowledge and handling the variability relations between them.

## 1.2 Contributions

This thesis brings the following contributions that support the thesis statement:

- **Approach to fully automated software product line evolution with diverse artifacts supported by a framework and tools (Chapter 3.6).**
- **Study of the complexity of in-code variability (Chapter 3.4).**
- **Establishing annotation-based software product lines as in-code variability observables of different complexity, size, and organization (Chapter 3.1).**
- **Matrix based approach to structural and semantic analysis supporting software product line evolution (Chapter 3.5).**
- **Approach to developing aspect-oriented software product lines with no aspects in products (Chapter 3.3).**
- **Approach to developing lightweight aspect-oriented software product lines with automated product derivation (Chapter 3.2).**

# 2. Handling Variability in Software Product Lines

Software product lines increase the effectiveness of product delivery by reusing existing assets, including code fragments, tests, and software artifacts. Handling variability represents the primary concern in software product lines. The primary consequence of introduced variability is getting beyond the single system development. Increased complexity has to be handled with respect to user requirements [34], interconnections between software components [1], and ensure as much reuse as possible. Consequently, variability handling has to be managed using different approaches, differentiating with the used technologies [16].

Variability management is necessary to select and integrate these assets as the main component of a software product line. The most commonly used implementation at the code level in the area of software product lines is conditional compilation [17]. Alternatively, a more comprehensive configuration and independence from the used programming language can be achieved with frame technology; specifically, marks/tags are used inside the source code, which becomes a template [30].

## 2.1 Implementing Variability in Code: Annotative Approaches

One of the first and largest software product lines are operating systems based on Linux kernel [24], where the problem with variability handling has emerged from the need to manage more than 12,000 features [42] across large 30 million lines of code from Linux operating system and its variants [29]. Already implemented features must be adapted to resolve conflicts with introduced functionality directly in code as part of software development. Conditional compilation based on C-preprocessor directives suits this purpose. Owing to this technology capable of including or excluding particular code fragments, many open-source [4] and industrial software product lines [48, 22] document high configurability [24]. Some of its known variants [4] are based on Linux kernel comprising FreeBSD [19], Fiasco [47], eCos [44], µClinux, and uClibc.

In the Linux kernel, its numerous components are reconfigured using even graphical user interfaces [4] into kernel images, which are perceived as final products to make them operate in respective scenarios demanding adaptations for specific platforms, subsystems, or even to take into account device drivers [6]. Despite the similarity with software product lines manifesting in the abovementioned high-configurability along with automated product generation, code reusability, no unrequested features, etc., no guidelines for product line scoping, feature modeling, development of core assets, etc. are applied in the development of Linux kernel [40]. These problems complicate perceiving them as software product lines.

Analyzing variability requires information about numerous features of these large systems. This information is preserved in variability models created using kind of variability model languages or even domain-specific languages [3], namely Kconfig [38] and Component Definition Language (CDL) [4]. These models—as well as similar feature models—belong to variability models. FreeBSD differentiates with the features organized in the list instead of in the hierarchical feature model known from other Linux kernels [39]. Additionally, FreeBSD counts only over 5000 features, which are more than doubled for eCos and Linux kernel according to the paper on reverse engineering of feature models from 2014 [39].

## 2.2 Implementing Variability in Code: Compositional Approaches

A complementary way of implementing variability in code to previously characterized annotative approaches lies in compositional approaches [15]. Aspect-oriented programming helps decompose features, enabling the capability to evolve features independently [37]. Plug-and-play adaptations of base behavior are ensured through the composition mechanism.

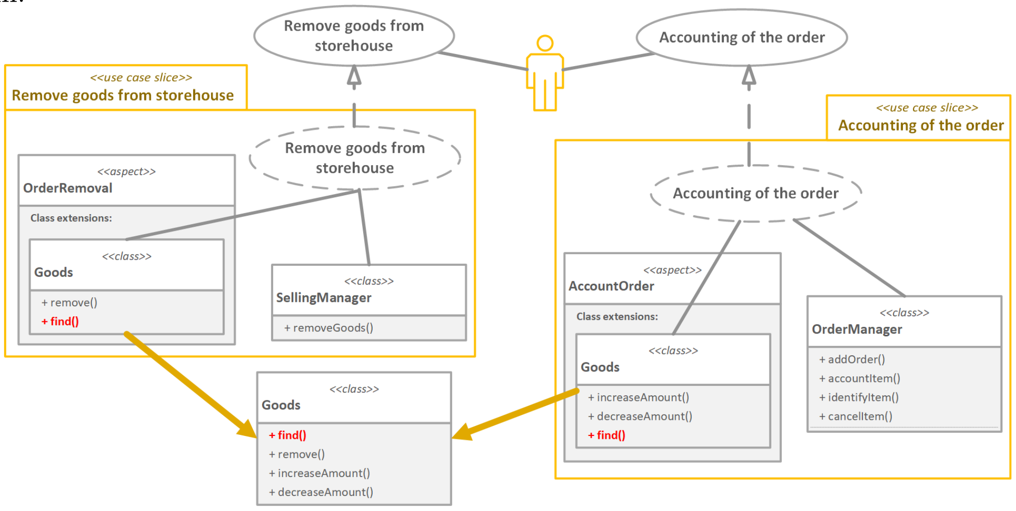

Figure 2.1: Modularization and composition of use cases (example adapted according to Michalco et al. [45])

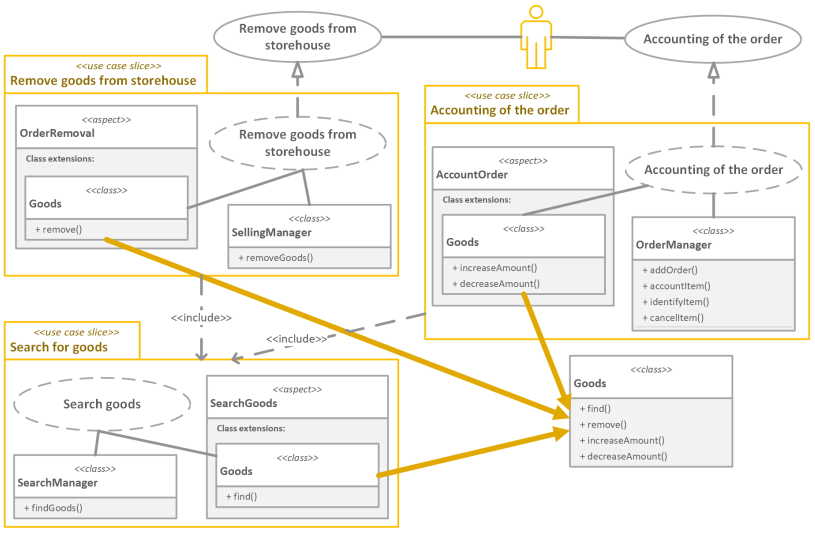

Figure 2.2: Resolving conflicts in use case modularization (example adapted according to Michalco et al. [45])

Feature-oriented approaches can modularize a feature into a single unit and abstract it from its application to the base definition of a particular variant. These abstracted deltas (such as calculating the price in the context of selling stock [31]) are expressed through mixins or subclasses. Mixin layers or abstract subclasses [7] are enhanced over mixin classes by high parameterization and the capability to capture collaboration on a multiclass level [41]. Mixins serve to implement collaborations by encapsulating other mixins [41], resulting in the creation of mixin layers. Despite their main benefits, they are restricted to modularizing nonhierarchical features [31].

Advanced decomposition owing to aspect-oriented programming also brings benefits to the specification and design phases by solving the use-case modularity problem [25]. The solution to this problem is required to represent *extends* and *includes* relationships using aspect-oriented constructs and propagate this information even into diagrams used in the specification and design phases of software development. Showing only information related to a particular use case or its parts, including common classes and crosscutting behavior with aspects, and only use-case-related methods and variables improve traceability. Consequently, this makes communication with customers about business functionality smooth because it concerns only a particular use case and nothing else. In the implementation phase, developers of these diagrams have direct information about system entities, their collaboration, and representation using object- and aspect-oriented programming. The demonstration with relation to the abovementioned symmetric aspect-oriented software composition based on peer use cases taking respective diagrams from Michalco's thesis [32] is shown in Figures 2.1 and 2.2. Use cases can be automatically replaced by more common themes for aspect-oriented programming [45]. The use-case decomposition is adaptable to incrementally establish software product lines from existing products, even at the design level. Variable features are represented and remain composed of whole using aspects. Similarly, use cases can then be set up together to tackle platform variation according to the incremental approach (designed by Alves et al. [2]) by resolving conflicts for different products (mobile games) according to particular constraints (phone devices).

## 2.3 Aspects in Products as a Limiting Factor

Aspects improve the flexibility and reconfigurability of software systems, mainly of product families, by controlling for the inclusion or exclusion of selected features [10]. The separation of concerns is achieved owing to the aspect orientation [43] used to compose or weave the functionality into variants. In addition, heterogeneous and homogeneous concerns can be managed in this manner. Compared with homogeneous concerns, heterogeneous concerns deal with different behavior at each control flow point [10]. Separating concerns also helps improve reusability by applying the same components in a different context, so they are not system-specific [36].

Despite these important benefits of aspects, many other problems emerged during their application. The use of aspect-oriented techniques requires understanding how join points are specified to determine and resolve conflicts between aspects and the modified or newly added source code that they influence [21]. Thus, aspects often complicate code debugging and can be error-prone. Another problem is their inability to influence many pieces of code inside a given method. The most advanced aspect-oriented languages, such as *AspectJ*, work on the method and attribute level [28]. To overcome this problem, hooks to hang the aspects are necessary [26]. The resulting products highly depend on aspects that address additional dependencies according to their nature, originating from the source implementation. For example, a special compiler is necessary for *AspectJ*, which requires consistent support updates in ever-evolving *Java*. Additionally, aspectual proxies cause load time-weaving problems [46] and their capabilities are provided at different levels of maturity [23], while the invasiveness of libraries in other languages causes problems during implementation. They even increase complexity owing to the unavailability of functionality for the modularization of components.

# 3. Methodology for Lightweight and Automated Evolution of In-Code Variability Supported by Frameworks

Challenges in handling variability appear after a high level of complexity is reached. At this point, an extensive number of features cannot be handled manually [9], particularly if features collide during their interaction. On the other hand, modularization using aspect-oriented programming tends to be hardly achievable [27]. Consequently, the features have to be evolved from the very beginning in code level to ensure extendability with optional employment of aspect-oriented programming and driven towards their massive introduction. Variability is flexibly handled in code using annotations leading to establishment of associated type of software product lines which are annotation-based. They are ensuring the establishment of highly configurable systems out of its features. Despite the benefits of annotation-based software product lines, in-code variability management is not adjusted to concisely preserve hierarchical information in the code in a lightweight, minimalistic, and fully automated manner. Specifically, annotations are left without a selection procedure based on in-code complexity, policy enforcement, naming conventions, managing domain knowledge, and criteria for maintaining feature models directly in the code. Consequently, we introduce a methodology for the establishment of a software product line in a lightweight fashion, a framework with competing strategies based on in-code complexities concerning the context of the entire code fragment to select the annotation with the lowest complexity, a scalable solution infrastructure by fast matrix-based methods for graph matching and clustering, and finally most of them entirely integrated into fully automated and minimalistic variability management. Additionally, the methodology is verified on established software product lines prepared in accordance with the variability managed in the code, but primarily on the minimalistic and fully automated version for managing fractals driven by structural information towards its semantic extensions.

## 3.1 Annotation-Based Software Product Lines

The resulting software product lines should be capable of efficiently studying interacting, stateful, and easily changeable feature combinations that can evolve from scratch fully automatedly. We ensured the mentioned capabilities with annotation-based software product lines characteristic with massive use of recursion and stateful dependency introduced by highly interactive UI elements bonded to the center element. Our lightweight aspect-oriented method, with its particular extensions and adaptations, is used as the technology to process annotations.

### 3.1.1 Software Product Line for Canvas-Based Applications

The variability in this stateful canvas-based software product line is managed across complex user interfaces, web services, components, and HTML templates. Each canvas element has predefined characteristics, such as position on the canvas, the way in which overlapping problems between elements are resolved, color, options to remove, duplicate, resize, crop, and others. Consequently, these manipulations predefine the states of the canvas and sometimes even have their own states used to handle the inner intermediate steps (sub-operations).

The *PuzzleToPlay* is a fully adapted environment designed to set up different types of puzzles to combine pieces to form the resulting shapes. The resized version is shown in Figure 3.1.

*Design3D* is an environment adapted to design the surfaces or textures of 3D objects intended to be used primarily on the web. The interface and majority of the available features are shown while designing the bottle cover, as shown in Figure 3.2.

### 3.1.2 Families of Fractal Products as a Source of Variability

Fractal products are much simpler because an application state does not span out of recursive behavior, making it manageable within the drawing. Each change is propagated into repetitive phases, causing the visual performance of the implemented feature to be infinitely detailed. Even minor changes in low code fragments tend to manifest as user-visible features owing to recursive behavior, allowing one to massively introduce new features and/or configure existing features and manage variability observable as infinitely detailed shapes.

Fractals are easily drawn inside the browser to the HTML canvas. Flexibility can be observed in the possibility of producing various data concerning multiple representations of the same fractal. Geometric shapes can be written as vectors or drawn as pixels in a raster image.

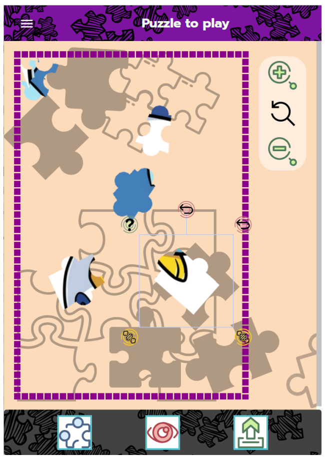

Figure 3.1: Preview of Puzzle To Play - resized

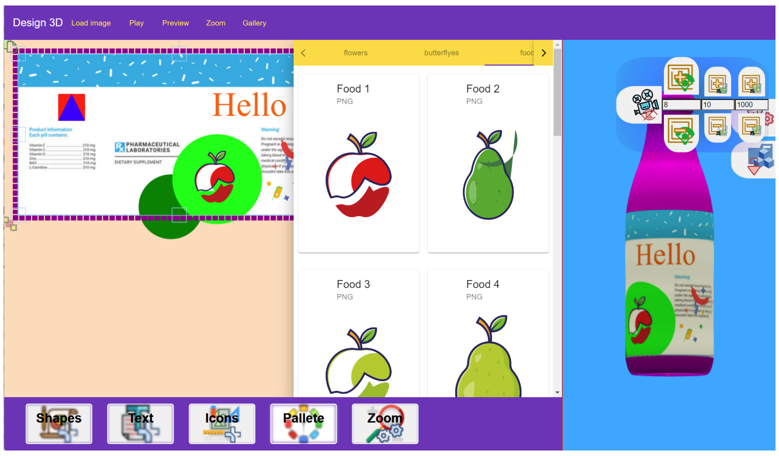

Figure 3.2: Preview of Design 3D

## 3.2 Lightweight Aspect-Oriented Software Product Lines with Automated Product Derivation

Establishing software product lines is challenging, particularly if they are based on a less widely accepted aspect-oriented paradigm. These two obstacles are addressed using an aspect-oriented approach capable of establishing software product lines with automated product derivation in our initial paper [33]. The easy setup is shown in fast application in two different cases, and the necessity to use only three types of annotations proved its lightweight nature. Firstly, annotations, as opposed to wrappers, are used in the form of comments. Secondly, the work introduces the hierarchical representation presented directly in JSON format in place of the variation points marked in the code. They are known as configuration expressions and are used to configure the inclusion or exclusion of an annotated code fragment in particular derived product. The complex rule is shown in Figure 3.3. These expressions even express relations of the feature models (feature trees) as shown in Figure 3.4 owing to their hierarchical nature and store additional information and knowledge, such as constraints and variability selection settings. This brings novelty to how variability, especially feature models, can be expressed and visualized in the code. Continuously, it opens space for further optimizations according to various complexity metrics and testing multiple design options to enhance variability comprehension of complex systems. Thirdly, fully automated product derivation is the main feature to highly benefit from reuse, decreased production costs, and other capabilities of software product lines.

```
{"AND": {
  "OR": {
     "computer": "false",
      "AND": {
         "computer": "true",
         "row": "topDown"
      }
   },
   playerNames": "true"
}
```

Figure 3.3: A complex hierarchical derivation rule and its capability to express feature models.

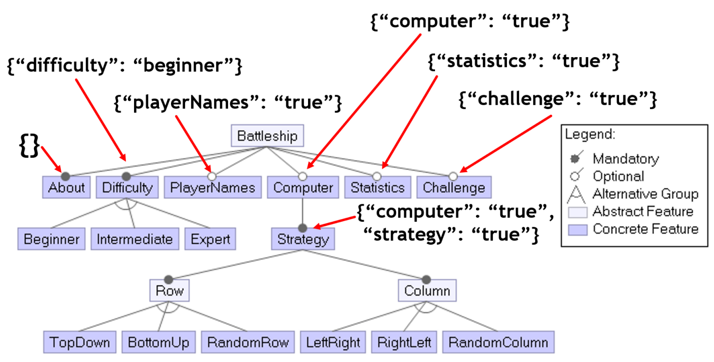

Figure 3.4: The feature model of Battleship game with derivation rules.

Annotations differ semantically according to their type, which prescribes how they should be recognized and handled by the functionality of the derivator. We propose three types of annotations:

**//@{}**
is an annotation used to manage the inclusion or exclusions of the entire file. This annotation can be applied to classes, aspects, and interfaces. In addition, it is feasible to configure the validation of some of their identification keywords when deriving such entities by the derivator.

**//#{}**
is annotation for functionality smaller than the size of particular file but still modular, especially to annotate functions. After evaluating the configuration expressions, such functionality will be included or excluded according to the truth value.

**//%{}**
is an annotation intended for one-line code fragments, particularly for import statements. This annotation must be used in combination with the previous annotation to support modularity as much as possible. Included functionality often introduces new code constructs that need to be imported.

## 3.3 Aspect-Oriented Software Product Lines with No Aspects in Products

Establishing a software product line that is capable of following the development practices prescribed by the chosen framework restricts the effective use of aspect-oriented techniques and tools. Primarily, composing components using prototype-based programming to evolve features independently cannot be performed or requires deviating from the framework for a single-page application and affecting development styles. Additionally, a new version of framework-based libraries prevents the incorporation of aspect-oriented libraries, rendering the introduced weaving mechanism nonfunctional. While following our previous approach, our focus on establishing it in a lightweight fashion resolves these problems by restricting the application of aspects only to variability management and bridging current development styles in single-page application development by completely removing aspects from the products. The direct focus on applying aspects exclusively to variability management brings the possibility of restricting and uniformly prescribing how each aspect is affected.

Consequently, we claim the ability to satisfy the contradictory consideration that aspects are unwanted for quantification and obliviousness, which are otherwise praised. We identified the advantages of software product lines and the possibility of adapting product derivation to satisfy their undesirability in products and incorporated them into a novel approach to establishing aspect-oriented software product lines without aspects in products supported by a framework. The lightweight fashion is introduced through *TypeScript* decorators, from which we benefit. Furthermore, it tackles how to make it simpler and more focused on the separation of variability following the idea of not propagating these variability decorators into the resulting products, ensuring aspect-free products. For this purpose, adaptations of design patterns are introduced to include binding their elements into code fragments from which variable features consist and making them fully detachable during product derivation, as can be seen in Figure 3.5 and Listing 3.1 that demonstrates it on the Decorator pattern. Only modular entities as variable code fragments performed at the code level can be annotated and easily detectable. The rest of the adaptations ensure the easy removal of aspect functionality from products while deriving fully functional products according to the configured features.

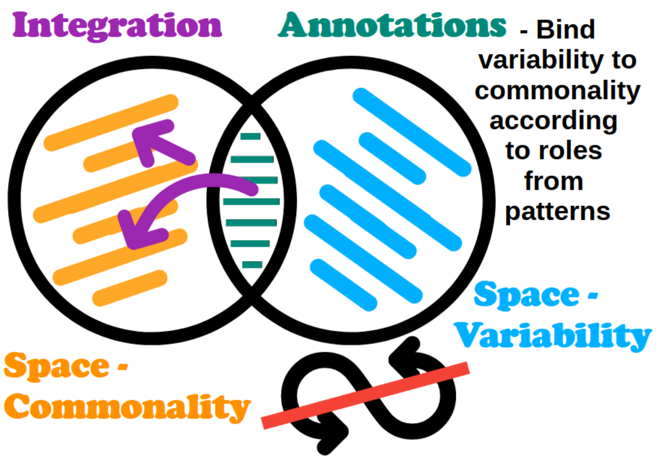
Integration
Annotations - Bind variability to commonality according to roles from patterns
Space - Variability
Space - Commonality

Figure 3.5: Variability handling after the introduction of adapted variability-detachable design patterns with their binding to code constructs

```
class Profile {
  rank: number = 4.5;
  @DecoratedFunc.
      incorporateMembershipRank({"
      group2": true})
  checkRank(threshold: number):
      boolean {
    return threshold < this.rank;
  }
}
```

Listing 3.1: Wrapped code of the method

## 3.4 Complexity of In-Code Variability: Emergence of Detachable Decorators

The constructs from known technologies for variability handling in code are mixed with business code such as preprocessor directives originating from conditional compilation or do not enforce policies when realized as comments observable in pure::variants or code is uncompilable using original compilers noticeable in employment of tags originating from frame technology and frame aspects. For example, Listing 3.2 taken from a demonstrative video presentation of commercial solution called pure:variants [35] shows the variability handling by wrapping the respective code fragment in comments. Their impact on code complexity with respect to similar approaches is left unnoticed and unmeasured.

Consequently, incorporating variability constructs inside the code according to our designed variability annotations (that overcome most of mentioned disadvantages and which is presented in Section 3.2) and wrapper based variants makes it possible to automatically compare them and even assess their complexity, especially the employment of feature models (hierarchical structures similar to trees) preserved in code. We are demonstrating it on example shown in Listing 3.3 which is pairwisely comparable with respective version from widely spread wrapper based variants in Listing 3.4.

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

Listing 3.2: Expressing in-code variability in pure::variants (adopted from pure::systems [35])

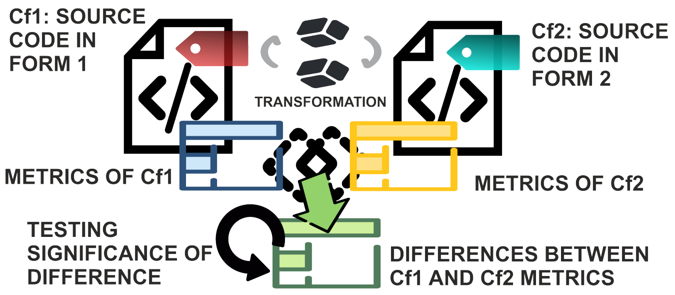
Cf1: SOURCE CODE IN FORM 1
TRANSFORMATION
Cf2: SOURCE CODE IN FORM 2
METRICS OF Cf1
METRICS OF Cf2
TESTING SIGNIFICANCE OF DIFFERENCE
DIFFERENCES BETWEEN Cf1 AND Cf2 METRICS

Figure 3.6: Comparative analysis of code constructs complexities of variability management

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

Listing 3.3: Annotated import statement by decorators

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

Listing 3.4: Wrapped code of the import statement

After extraction and optional updates, their complexity can be measured in a particular code context. Consequently, we designed the way how to compare and evaluate code constructs of variability management. The key is the visualization of the code, its complexity assessments, and their differences in the context of an analyzed modular unit consisting of the analyzed construct and surrounding code. Subsequently, the pair of code complexities from the state before and after the particular transformation of each file in the entire software product line are statistically tested to evaluate the consequences of incorporating different variability constructs. This paired statistical test provides evidence of a significant difference between the complexity of the initial and transformed codes across the entire software product line. The characteristic scheme, including compared scripts after their transformation into predefined forms, evaluated metrics with their differences, and statistical evaluation under the collected pairs across the entire software product line, is drawn in Figure 3.6.

## 3.5 Matrix Based Approach to Structural and Semantic Analysis for Software Product Line Evolution

The creation of feature models (feature trees) using hierarchical clustering [8] is not a new idea, and it has been successfully used in software product lines for some time, especially in Aspect-oriented product line engineering (AMPLE) project [37]. Initially, the application of their clustering algorithms consists of mining features with their dependencies, usually with latent semantic analysis primarily from documentation and code. We proceed further when we cover not only semantics but even structural information and the entire range between them to produce resoective views in fully automated fashion. In its application we benefit from existing modular units of software applications and do not directly focus on feature extraction. In our case, we match similar nodes using fast polynomial algorithms with the ability to characterize the similarity of edges (between software components) optionally and consider node importance in accordance with dependencies on other modules. The significant difference lies in its ability to be applied in a fully automated fashion as a primary element in the Big Data infrastructure. We intend to support views on existing variability-driven software product line semantics, structure, and further automated domain knowledge extraction.

we consider the existing modular units of software applications and do not directly focus on feature extraction. Instead, Entire incorporation of matrix based matching and clustering algorithms into a configurable approach (extended with more complex edge-similarity scores) to support software product line evolution is shown in the flowchart diagram in Figure 3.7.

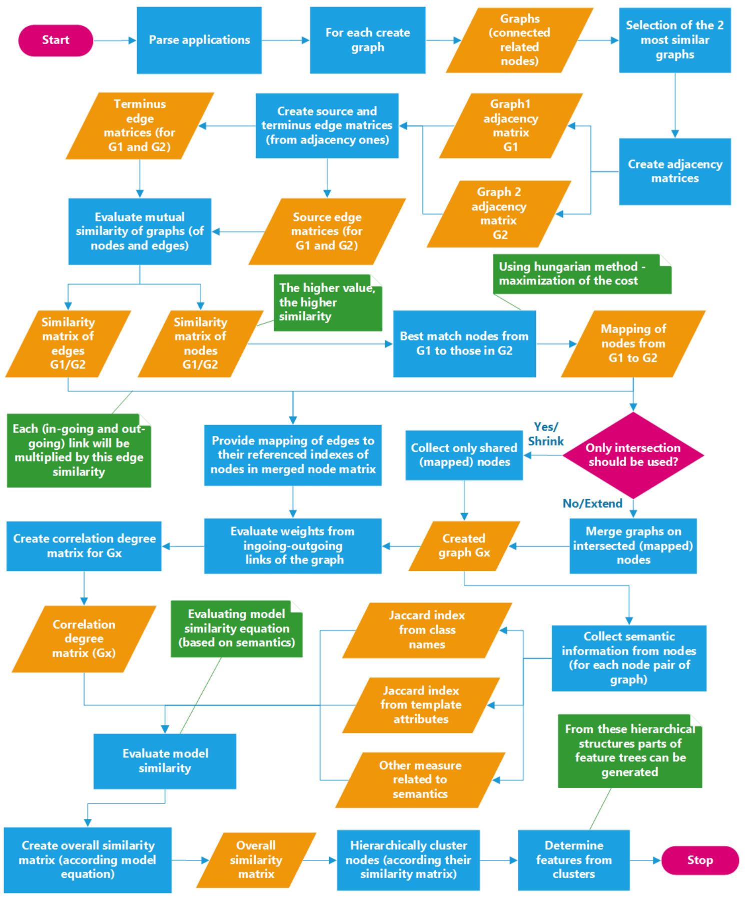
Start
Parse applications
For each create graph
Graphs (connected related nodes)
Selection of the 2 most similar graphs
Terminus edge matrices (for G1 and G2)
Create source and terminus edge matrices (from adjacency ones)
Graph1 adjacency matrix G1
Create adjacency matrices
Graph 2 adjacency matrix G2
Evaluate mutual similarity of graphs (of nodes and edges)
Source edge matrices (for G1 and G2)
Using hungarian method - maximization of the cost
The higher value, the higher similarity
Similarity matrix of edges G1/G2
Similarity matrix of nodes G1/G2
Best match nodes from G1 to those in G2
Mapping of nodes from G1 to G2
Each (in-going and out-going) link will be multiplied by this edge similarity
Provide mapping of edges to their referenced indexes of nodes in merged node matrix
Collect only shared (mapped) nodes
Yes/ Shrink
Only intersection should be used?
No/Extend
Create correlation degree matrix for Gx
Evaluate weights from ingoing-outgoing links of the graph
Created graph Gx
Merge graphs on intersected (mapped) nodes
Correlation degree matrix (Gx)
Evaluating model similarity equation (based on semantics)
Jaccard index from class names
Collect semantic information from nodes (for each node pair of graph)
Jaccard index from template attributes
Evaluate model similarity
From these hierarchical structures parts of feature trees can be generated
Other measure related to semantics
Create overall similarity matrix (according model equation)
Overall similarity matrix
Hierarchically cluster nodes (according their similarity matrix)
Determine features from clusters
Stop

Figure 3.7: Matrix based semantic analysis of features enhanced with node scores

## 3.6 Fully Automated Software Product Line Evolution with Diverse Artifacts

Existing approaches in software product lines usually neglect knowledge modeling and simulation of the interaction between features capable of bringing dynamism and automation. Consequently, these solutions miss opportunities to resolve associated and emerging problems, including defect detection or quality assurance, which can be solved by effectively extracting and utilizing knowledge from data based on the differences between variants. We bring capabilities to seize them in introducing a fully automated and minimalistic approach to software product line evolution that strictly focuses on handling variability at low-level code fragments.

The minimalistic evolution concerning variability handling in the program is handled in the processed abstract syntax tree and then in the code according to the following points:

**Configuring parameters**

paramVP[id] Variability is expressed using wrappers

**Annotating negative variability**

AnnotationVP[id] Variability is expressed using detachable decorators but without variability configuration expressions

**Marking positive variability**

markerVP[id] Variability is expressed using detachable decorators

It incorporates the autonomous modeling of emerging knowledge across preconfigured simulations. Specifically, fully automated knowledge-driven software product line evolution provides various views on an existing software product line, its variants, and their evolution through semantic and structural information accompanied by the time and order of the performed changes. We initially developed our approach for software product line evolution, followed by its successful application to the evolution of fractal scripts. The emerging instance of the five-sided fractal software product line and its possibly derived products must be analyzed and abstracted from particular information in multiple ways, each being treated as a product-representative view.

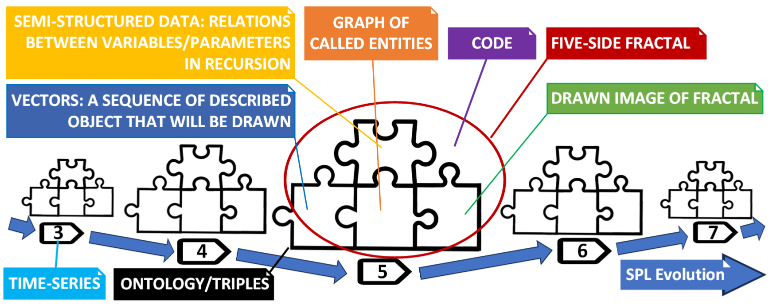
SEMI-STRUCTURED DATA: RELATIONS BETWEEN VARIABLES/PARAMETERS IN RECURSION
GRAPH OF CALLED ENTITIES
CODE
FIVE-SIDE FRACTAL
VECTORS: A SEQUENCE OF DESCRIBED OBJECT THAT WILL BE DRAWN
DRAWN IMAGE OF FRACTAL
3
4
5
6
7
TIME-SERIES
ONTOLOGY/TRIPLES
SPL Evolution

Figure 3.8: Relations amongst diverse representations for presented five-sided fractal in software product line evolution

The whole fractal is effectively captured in the respective views presented as puzzles (code from which is solution set up) that are combined into a whole using ontology, as shown in Figure 3.8.

The fully automated evolution applied to evolution of fractals is visualized in Figure 3.9 and can be compared even to its respective generalization shown in Figure 3.10.

Fractal products are much simpler because an application state does not span out of recursive behavior, making it manageable within the drawing. Each change is propagated into repetitive phases, causing the visual performance of the implemented feature to be infinitely detailed. Even minor changes in low code fragments tend to manifest as user-visible features owing to recursive behavior, allowing one to massively introduce new features and/or configure existing features and manage variability observable as infinitely detailed shapes.

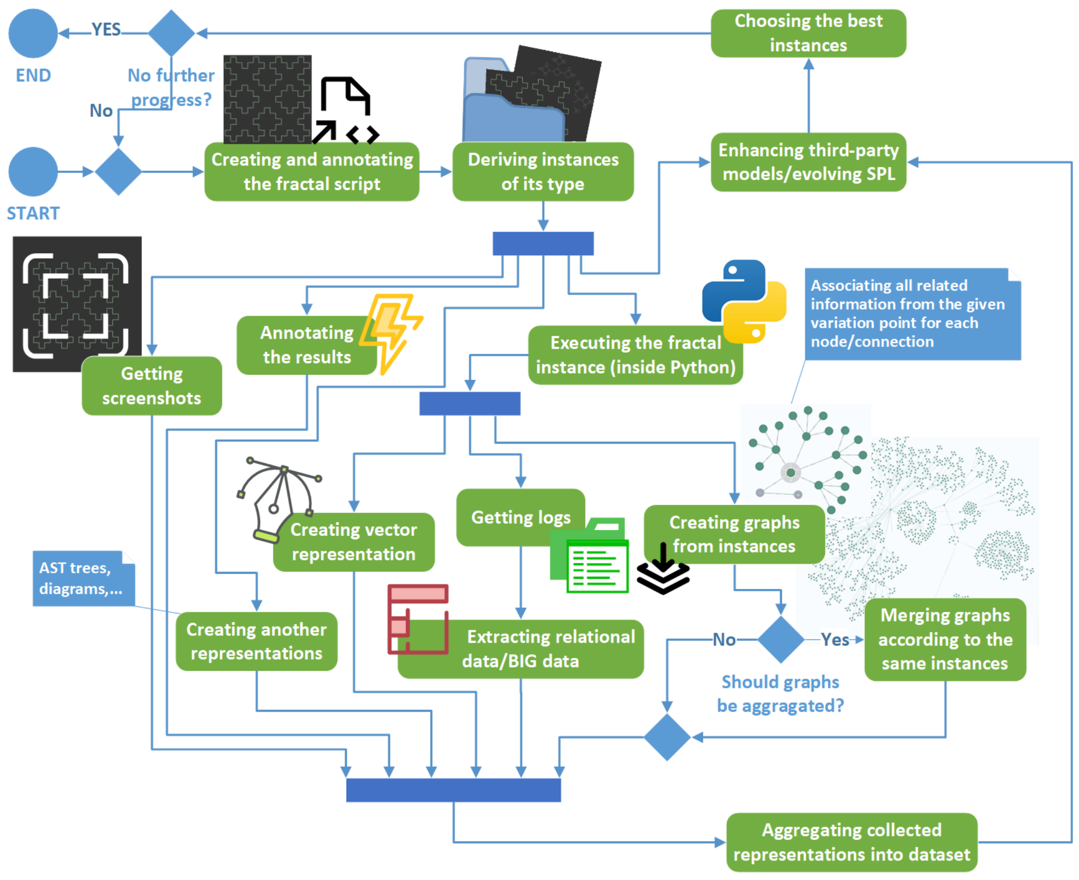
END
YES
No further progress?
No
START
Choosing the best instances
Creating and annotating the fractal script
Deriving instances of its type
Enhancing third-party models/evolving SPL
Annotating the results
Executing the fractal instance (inside Python)
Associating all related information from the given variation point for each node/connection
Getting screenshots
Creating vector representation
Getting logs
Creating graphs from instances
AST trees, diagrams,...
Creating another representations
Extracting relational data/BIG data
No
Yes
Merging graphs according to the same instances
Should graphs be aggregated?
Aggregating collected representations into dataset

Figure 3.9: The process of getting various representations from the fractal script and using it for the software product line evolution

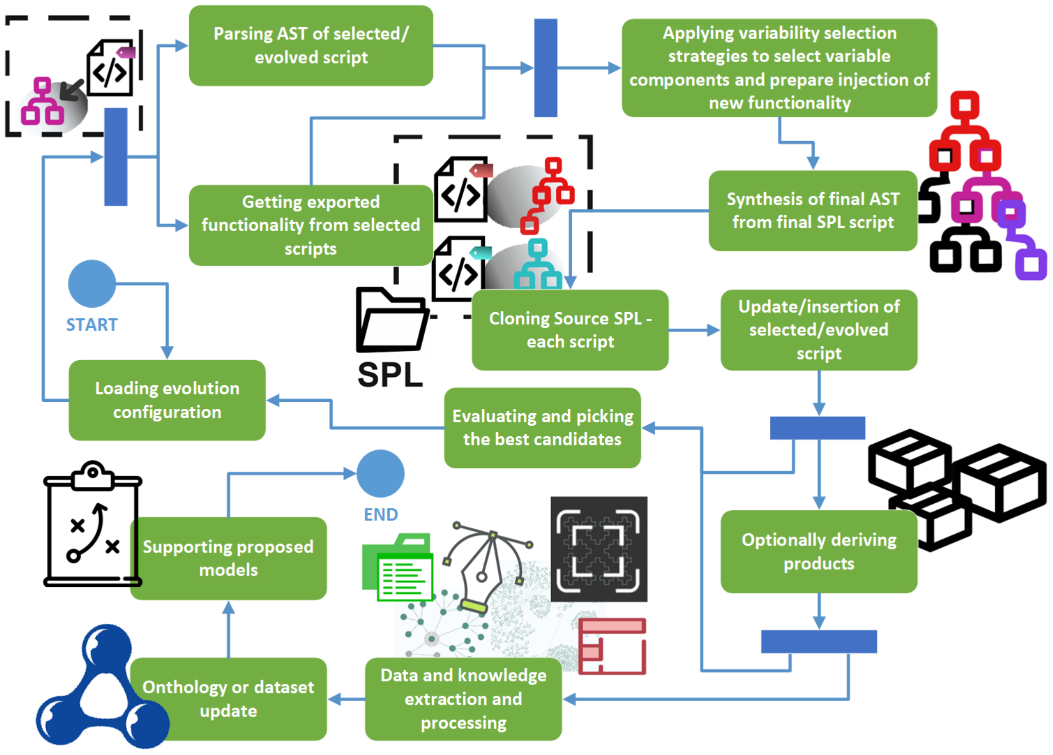
Parsing AST of selected/ evolved script
Applying variability selection strategies to select variable components and prepare injection of new functionality
Getting exported functionality from selected scripts
Synthesis of final AST from final SPL script
START
SPL
Cloning Source SPL - each script
Update/insertion of selected/evolved script
Loading evolution configuration
Evaluating and picking the best candidates
END
Optionally deriving products
Supporting proposed models
Onthology or dataset update
Data and knowledge extraction and processing

Figure 3.10: High-level view on software product line evolution process

# 4. Results

This thesis brings the following contributions that support the thesis statement:

- **Approach to fully automated software product line evolution with diverse artifacts supported by a framework and tools (Chapter 3.6).** The approach is driven by the iterative and incremental establishment of features based on structural and semantic information within the code. We support it with the implementation of the framework. The minimalism we achieved by focusing only on variability handling which consists of annotating features as common or variable and introducing features automatically by inserting calls to existing functionality according to the contextual information, including calls that can be made using available variables. Diverse representations originating from variability annotated source code are extracted and organized in a semantic graph to derive new knowledge and models for making decisions about evolution based on semantics. The other tool automatically records part of the domain knowledge contained within the variability configurations.

- **Study of the complexity of in-code variability (Chapter 3.4).** This study justifies the in-code representation of annotations used in our initial lightweight aspect-oriented approach with automated product derivation. We introduced an automated approach to an exploratory study of the in-code complexity of constructs belonging to variability management, which primarily enables the comparison of annotations adapted for variability handling using code complexity metrics. The approach even supports the optional design of new constructs used for the variability handling in code. Practical insights into the usability of new code constructs can be analyzed, such as the capability to not blend with the rest of the business logic or to capture a particular place in the code visually. We supported the approach to the exploratory study of variability management in-code complexity with a framework to evaluate complexities automatically. We statistically and automatically compared detachable decorators with wrapper constructs (in-code representation of tags from frame technology or block comments from pure::variants). From this evaluation, we determined that detachable decorators are significantly less complex. From an evaluation based on comparing complexity with and without any variability constructs, we gained insights into how determined representation with low complexity can be optimized further with information about the threshold obtained from each complexity metric. We recommend supporting some illegal decorators/annotations according to statistical evaluation that, on the borderline, proved the significance difference between complexity caused by the necessity of positioning dead code constructs visually next to the code fragments belonging to variability and without these dead code constructs (the code is compared with and without these dead code constructs).

- **Establishing annotation-based software product lines as in-code variability observables of different complexity, size, and organization (Chapter 3.1).** Various aspects of in-code variability management have been neglected in existing software product lines, including evolution based on structural information from the code, framework restrictions, immature aspects to capture constructors or be chained, and merging platforms. The introduced single-page applications and established software product lines are perceived as tools for handling various challenges related to variability management. We developed two highly differentiating canvas-based *Angular* single-page applications called *Puzzle to Play* and *Design 3D* that can be supplied for commercial purposes to be used as sources in the establishment of the respective software product line, followed by the possibility of analyzing the problems that occurred in their integration. From these applications, we established a software product line for canvas-based single-page applications through which we achieved high usability, statefulness, and reuse. However, we tackled and resolved issues with variability handling related to the restrictions of the *Angular* framework and the applicability of aspect-oriented programming. Similarly, we introduced and established a software product line for evolving fractals by following and successfully fulfilling the aim to handle variability in small code scripts. We achieved the feasibility of merging two platforms from introduced software product lines into one, which leads to the introduction of new kinds of products owing to the possibility of intertwining the software product line to evolve fractal scripts with those for the evolution of canvas-based applications and analyzing the transition from software product line evolution based on structural information to those demanding comprehension of semantic information to adapt change into particular (even stateful) context.

- **Matrix based approach to structural and semantic analysis supporting software product line evolution (Chapter 3.5).** We integrated fast hybrid methods for graph matching and clustering operating using similarity metrics into a proposed fast approach capable of massively merging graphs and clustering them into structural and semantic views to support software product line evolution. We supported the decisions to merge related nodes or omit them performed on the graph with the additionally evaluated similarities between the connections. We supported our approach with a tool to persist and visualize graphs in a graph database. An adjacency matrix is extracted by retrieving the associated semantic information for a particular graph using the respective graph queries. *Angular* applications are analyzed as modular units perceived as software product line components by observing their relatedness and characterizing them using a similarity measure. Subsequently, we produce semantic and structural views by balancing this measure.

- **Approach to developing aspect-oriented software product lines with no aspects in products (Chapter 3.3).** We transformed the three types of annotations introduced in our approach, called lightweight software product lines with fully automated product derivation, into code constructs based on decorators to handle variability exclusively. We developed an approach to establish a software product line for *Angular* canvas-based single-page applications based on these annotations. We propose aspect-free products as a solution to issues with immature aspects incapable of capturing constructor calls or being chained and restrictions introduced by the *Angular* framework, especially the necessity to register components in modules that prevent the advanced modularization of features using feature-oriented decomposition. To ensure product derivation, we prepared a product derivator tool capable of removing aspects from final products using adaptations of recreated object-oriented design patterns into aspect orientation for easily detachable components belonging to the variability without leaving aspects in code. In its design, we ensured minimalism and ease of removing aspects from resulting products by applying aspects exclusively to implement variability management. In addition, we proposed a new annotation type to easily copy the code to the requested location according to the variability configuration.

- **Approach to developing lightweight aspect-oriented software product lines with automated product derivation (Chapter 3.2).** To establish lightweight aspect-oriented software product lines, we overcome two primary problems: establishing a product line is not trivial, and aspects are not widely spread. We introduced an approach to establish a lightweight aspect-oriented software product line with three designed types of annotations incorporated using comments in the code to annotate variability in code and a custom program to derive final products in an automated manner. In this approach, we proposed concise hierarchical configuration expressions in JSON format inside our annotations, following our initial idea of preserving feature models (feature trees) in code with perspectives of advantages of its applicability in case of full automation and improving comprehension of polluted source code with annotations from annotation-based software product lines.

## 4.1 Clarifying Lightweight Nature

To evaluate impact of handled variability in code we set up following hypotheses:

**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.

Table 4.1: 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 Identifiers Distinct Operands | 0.9996 | 2415 | 2.5713E-16 | 2.00, 2.00 | 2.0000 | **FALSE** |
| Halstead's Identifiers Total Operands | 0.9999 | 2556 | 9.3382E-17 | 2.00, 2.00 | 2.0000 | **FALSE** |
| Halstead's Identifiers Distinct Operators | 0.9886 | 2030 | 2.1410E-12 | 1.00, 1.50 | 1.0000 | **FALSE** |
| Halstead's Identifiers Total 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 first case concerns annotating the variability using detachable decorators. The decorator pattern is applied to hang annotations related to the variability handling of specific constructs in code, such as methods, classes, or declared variables, and enforced separation of these constructs and their effects from code and the use of particular names to differentiate them. In addition, their dynamic updates are directly feasible from the code. This in-code representation results in the modular preservation of code fragments belonging to the variability in classes and methods designed to be performed in an automated fashion.

The second case concisely contains information about the complexity of configuration expressions in the JSON format by evaluating them as *JavaScript/TypeScript* objects. We achieved the capability to optimize these expressions according to the context of the code, primarily by using code complexity measures.

The third case is based on traditional wrapper constructs, and in comparison with the first case, it resulted in higher complexity in the 15 used code complexity metrics, including LOC, Halstead's measures, cyclomatic complexity, and cyclomatic density. Our wrappers, represented by variables bounding the code from the top and bottom, solved this problem by evaluating the in-code complexity of the comments used in pure::variants. The results of statistical Wilcoxon test are presented in Table 4.1.

The fourth case contains no variability handling constructs to gain a baseline for code complexity optimization when searching for low-complexity constructs. It helps classify emerged variability-handling constructs according to their contribution to the additional complexity and evaluate preferences in optimizing such complexity towards its possible insignificance.

The last case concerns the complexity of the annotated dead code constructs, which are visually positioned next to the code fragments that cannot be decorated. The results showed a significance value near the threshold but still resulted in a significant complexity overhead for the majority of metrics. It pointed to the effects of legalizing illegal decorators for one-line constructs, such as import statements, while producing less complex codes. Less complexity is achieved by replacing the wrappers with detachable decorators.

## 4.2 Universal In-Code Variability Handling: High Level Synergy of Methodological Contributions

The combination of approaches through which we reached synergy leads to establishing software product lines concerning variability handling in a lightweight, complexity-optimized way, with applications to minimalistic and fully automated evolution of software product lines, extendability, preserving feature models in code, and many applications as follows.

The composition of all the approaches presented in this thesis is organized in a sequence of questions related to in-code variability handling to choose and apply the corresponding steps with the activity diagram shown in Figure 4.1. Initially, the in-code constructs to preserve particular information in code while considering their application must be discovered using the approach supported by the framework from our study called *Complexity of In-Code Variability: Emergence of Detachable Decorators (Chapter 3.4)*. According to this study, when we handle variability in annotation-based software product lines, decorators emerge as the construct with the lowest complexity, are detachable, and are not mixed with business code, but other types can be reconsidered in this step. Subsequently, the following question considers an analytical aim to observe and adapt changes to components from the beginning of the automated software product line evolution. For example, suppose we want to optimize the configuration expressions or restructure the code of the annotation-based software product line during iterative and incremental incorporation of changes. In this case, it is possible to use our approach, called *Fully Automated Software Product Line Evolution with Diverse Artifacts (Chapter 3.6)*, applied to the evolution of fractal scripts. In contrast, our aim is to establish or extend a software product line with a new software application (product) instead.

Additionally, it should be possible to evolve software product lines to fulfill future business goals. When the answer to these questions is positive, then our *Matrix Based Approach to Structural and Semantic Analysis Supporting Software Product Line Evolution (Chapter 3.5)* must be used to analyze intersections of common and variable features from the software applications to establish software product line or from the existing software product line and software application to properly extend this product line. Whether the previously applied approach is applied or not, the following question concerns the complexity of variability annotated scripts in an annotation-based software product line.

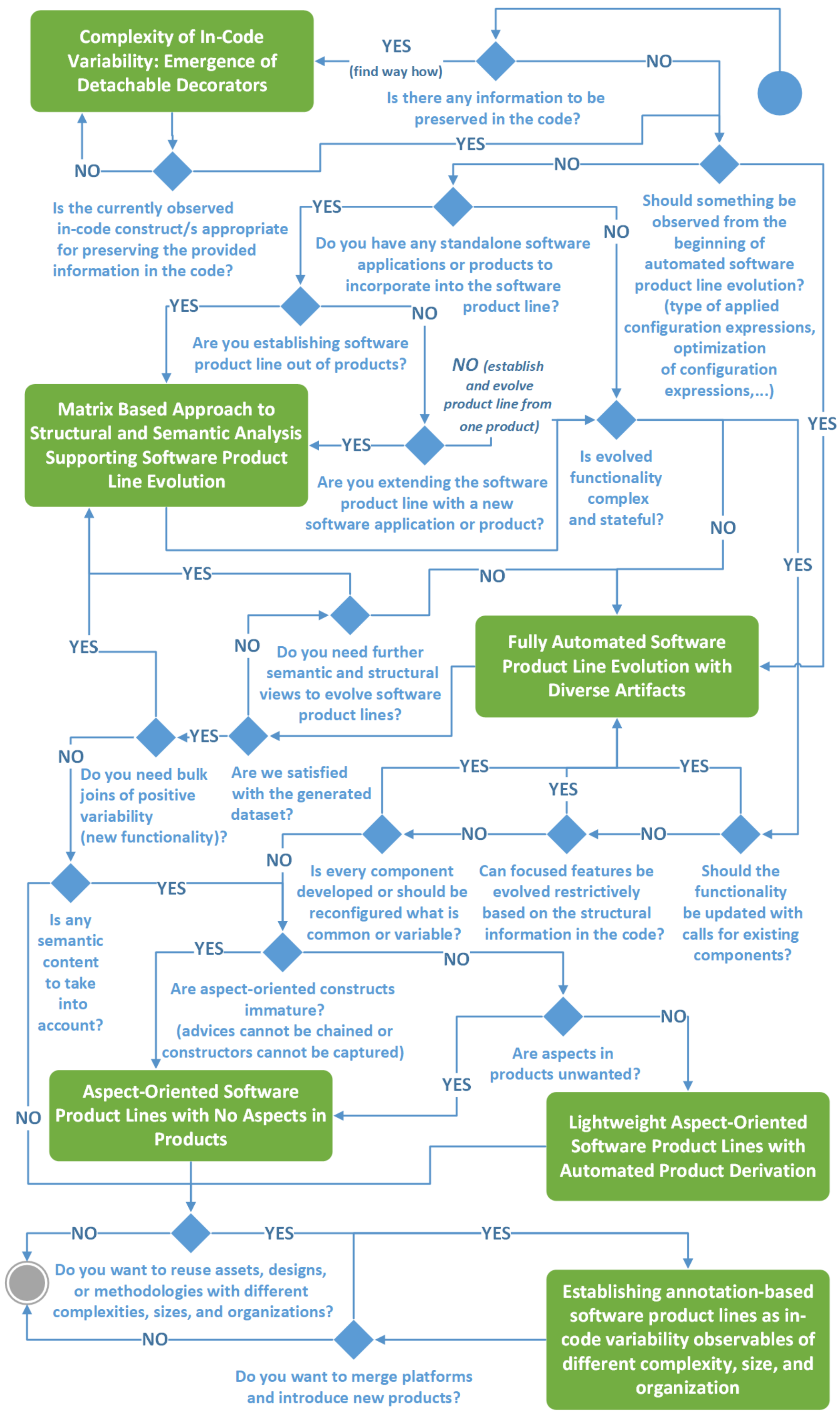
Complexity of In-Code Variability: Emergence of Detachable Decorators
YES (find way how)
NO
Is there any information to be preserved in the code?
YES
NO
Is the currently observed in-code construct/s appropriate for preserving the provided information in the code?
YES
Do you have any standalone software applications or products to incorporate into the software product line?
NO
Should something be observed from the beginning of automated software product line evolution? (type of applied configuration expressions, optimization of configuration expressions,...)
YES
NO
Are you establishing software product line out of products?
NO (establish and evolve product line from one product)
Matrix Based Approach to Structural and Semantic Analysis Supporting Software Product Line Evolution
YES
Are you extending the software product line with a new software application or product?
Is evolved functionality complex and stateful?
YES
NO
YES
NO
Do you need further semantic and structural views to evolve software product lines?
Fully Automated Software Product Line Evolution with Diverse Artifacts
YES
NO
Do you need bulk joins of positive variability (new functionality)?
YES
Are we satisfied with the generated dataset?
YES
YES
YES
NO
NO
Is every component developed or should be reconfigured what is common or variable?
Can focused features be evolved restrictively based on the structural information in the code?
Should the functionality be updated with calls for existing components?
YES
Is any semantic content to take into account?
YES
NO
Are aspect-oriented constructs immature? (advices cannot be chained or constructors cannot be captured)
NO
Are aspects in products unwanted?
YES
Aspect-Oriented Software Product Lines with No Aspects in Products
NO
Lightweight Aspect-Oriented Software Product Lines with Automated Product Derivation
NO
YES
YES
Do you want to reuse assets, designs, or methodologies with different complexities, sizes, and organizations?
Establishing annotation-based software product lines as in-code variability observables of different complexity, size, and organization
NO
Do you want to merge platforms and introduce new products?

Figure 4.1: Evolving software product lines with in-code variability in a lightweight and automated manner

Suppose that functionality (software product line scripts with or without annotated variability) is not complex and stateful. In this case, it can evolve using our *Fully Automated Software Product Line Evolution with Diverse Artifacts (Chapter 3.6)*. Otherwise, the cases must be checked to see if the previously mentioned approach to software product line evolution cannot be applied. For example, if the existing complex functionality has already been implemented, it can only be called from a particular location handled during such automated evolution. Another case is when we evolve something based on structural information, such as recursively affecting executed functions or extending existing entities with new variables and methods. The last case occurs if what is common and the variable must be specified or reconsidered. When we are not doing any of these updates (our answer is NO for all three of these questions), we must establish a software product line using a different approach.

Consequently, the benefits of advanced modularization and the associated dependencies on aspect-oriented programming must be considered. When aspects as part of the aspect-oriented paradigm are immature, which manifests in the inability to chain the advices around a particular variation point, capture class constructor calls, etc., we should use our *Aspect-Oriented Software Product Lines with No Aspects in Products (Chapter 3.3)* approach. Otherwise, we must check whether the aspects are unwanted for some reason. The primary reason can be caused by problems with the additional modularization of concerns when the complexity of the resulting solution rapidly increases [14, 27]. The aspects that affect a particular place cannot be forgotten even if they are not defined in that place. Accordingly, changes in software development practices and skills in aspect-oriented programming are necessary. In addition, the outdated aspect-oriented *AspectJ* compiler with the current *Java* version complicates the incorporation of new language features. When we accept aspect-oriented programming with undetachable dependencies, we should proceed by establishing a software product line using our lightweight and semi-automated approach called *Lightweight Aspect-Oriented Software Product Lines with Automated Product Derivation (Chapter 3.2).*

Software product line evolution based on the *Fully Automated Software Product Line Evolution with Diverse Artifacts (Chapter 3.6)* approach can be repeatedly applied. This repetitive application can occur if one is unsatisfied with a generated dataset from diverse representations, typically associated with the outcomes of evolution. Consequently, semantic and structural views are necessary to obtain an overview of suitable changes to benefit from the evolution of the software product line. In this case, we obtain these views using our *Matrix Based Approach to Structural and Semantic Analysis Supporting Software Product Line Evolution (Chapter 3.5)* and continue as mentioned above or according to the sequence diagram shown in Figure 4.1. Otherwise, we repeat our *Fully Automated Software Product Line Evolution with Diverse Artifacts (Chapter 3.6).* If we incorporate more features simultaneously in this type of evolution, then it is suitable to inject functionality using bulk joins. Because this process operates under graphs made from dynamically instantiated entities, the *Matrix Based Approach to Structural and Semantic Analysis Supporting Software Product Line Evolution (Chapter 3.5)* easily merges these graphs according to semantic or structural similarities. When it is necessary to proceed by evolving based on semantic information, it is beneficial to select the most appropriate approach (*Lightweight Aspect-Oriented Software Product Lines with Automated Product Derivation (Chapter 3.2)* or *Aspect-Oriented Software Product Lines with No Aspects in Products (Chapter 3.3)*) according to the aforementioned decisions regarding the incorporation of aspect-oriented programming.

In the last phase, we must decide on reusing assets, designs, or methodologies from the established software product line. When we proceed, we extend our observable software product lines that are primarily perceived as tools for variability handling presented in *Chapter 3.1*, called *Establishing annotation-based software product lines as in-code variability observables of different complexity, size, and organization.* These tools allow us to reuse and combine assets, designs, and methodologies until they fulfill our goals. Finally, we successfully evolved an annotation-based software product line in the code, followed by the termination of the entire process.

# 5. Conclusions

Annotation-based software product lines are widely spread software product lines characterized by managing variability in code. Their prominent representatives, Linux kernel-based operating systems, are the largest in the number of features, counting over several thousand of them. Despite their benefits, several approaches and paradigms, including aspect-oriented programming and feature-oriented decomposition, overcome them when successfully separating and modularizing features. They demand additional tool support and changes in the development practices and paradigms. Primarily, unknown effects from the outside to a particular code fragment known as obliviousness [18] at a specific place in the code complicate their applicability in a lightweight manner. Following this perspective, the effects on user cognitive load demand the analysis of complexity and ease of comprehending variability annotations in code.

The current state of the art lacks the ground to perform this operation because of overlapping annotated variability and business logic in the annotation-based version of software product lines. However, other versions following feature-oriented decomposition do not exhibit this problem because they have fully separated variability. However, they even make code oblivious to incorporated changes with the necessity to track these changes. Automated evolution is more complicated and unrealistic for some changes owing to increased complexity. Accordingly, we perceive that the ground for full automation is left undiscovered in the context of the lightweight establishment of software product lines. Specifically, it is necessary to produce quality products massively and tackle the increased complexity caused by the introduced variability. After that, advanced techniques will be employed including aspect-oriented programming.To the best of our knowledge, no research has focused on creation/generating and annotating features from the early beginning in an automated fashion to gain insights into the domain. Consequently, we designed a methodology to bring full automation into software product line evolution encompassing diverse artifacts to organize knowledge from modeling and simulations of variability contained in the code so that all of these operations are performed in a minimalistic and lightweight fashion.

The clarification of performed design decisions for such in-code variability management consists of benefiting from achieved minimalism, preservation of hierarchic information in code, and optimizations of annotations for variability handling as drivers of our fully automated evolution. Minimalism lies in focusing restrictively on the variability handling that is based on our proposed lightweight complexity-optimized approach for establishing annotation-based software product lines with variability handling in code. Ability to preserve hierarchical information in code will improve comprehensibility through preserving feature models in code. Used annotations are compared and selected based on lower in-code complexity while supporting policy enforcement, naming conventions, managing domain knowledge, and observing criteria for maintaining feature models directly in the code, which will improve the comprehensibility of the variability managed in the code. Systematic and iterative establishment of features in software product lines can be achieved based on structural information taken from source code by subsequently applying it to discover domain knowledge through wiring diverse representations originating from variability annotated source code into a semantic graph, providing the capability to handle variability from the beginning fully automatically.

# Bibliography

[1] N. Abbas and J. Andersson. Harnessing variability in product-lines of self-adaptive software systems. In *Proceedings of the 19th International Conference on Software Product Line*, pages 191–200, Nashville Tennessee, 07 2015. ACM. doi: 10.1145/2791060.2791089.

[2] V. Alves, P. M. Jr, and P. Borba. An incremental aspect-oriented product line method for j2me game development. In *Workshop on Managing Variability Consistently in Design and Code (in conjunction with OOPSLA ´2004)*, page 3, 01 2004.

[3] T. Berger. variability, 2011. URL `https://code.google.com/archive/p/variability/`.

[4] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *Software Engineering, IEEE Transactions on*, 39:1611–1640, 12 2013.

[5] J. Bosch. *Design & Use of Software Architectures—Adopting and Evolving a Product Line Approach.* Addison-Wesley Professional, 01 2000. ISBN 978-0-201-67494-1.

[6] D. Bovet and M. Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[7] G. Bracha and W. Cook. Mixin-based inheritance. *SIGPLAN Not.*, 25(10):303–311, Sept. 1990.

[8] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 31–40, 2005.

[9] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90, 01 2009.

[10] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical report, Lancaster University, 2004. vol 107.

[11] H. M. Company. *The American Heritage Dictionary of the English Language, Fourth Edition.* Houghton Mifflin Company, 4 edition, 2009.

[12] K. Czarnecki. *Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Department of Computer Science and Automation: Technical University of Ilmenau, 1999.

[13] T. M. Dao and K. C. Kang. Mapping Features to Reusable Components: A Problem Frames-Based Approach. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen,

M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Bosch, and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287, pages 377–392. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] I. Despi and L. Luca. Aspect oriented programming challenges. *Anale. Seria Informatică, 2(1)*, 2004. URL `https://hdl.handle.net/1959.11/7998`.

[15] S. El-Sharkawy, N. Yamagishi-Eichler, and K. Schmid. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, 106:1–30, 2019.

[16] W. Fenske, T. Thüm, and G. Saake. A taxonomy of software product line reengineering. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 1–8, Sophia Antipolis France, 01 2014. ACM.

[17] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of Proceedings of the 30th international conference on Software engineerin, ICSE'08*. ACM, 2008.

[18] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000*, page 9, Minneapolis, Minnesota USA, 2000. RIACS. "RIACS Technical Report 01.12, 2001".

[19] FreeBSD Foundation. The freebsd project, 2025. URL `https://www.freebsd.org/`.

[20] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems—a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014.

[21] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*, page 18, 2003.

[22] R. Hellebrand, A. Silva, M. Becker, B. Zhang, K. Sierszecki, and J. Savolainen. Coevolution of variability models and code: An industrial case study. In *SPLC '14: 18th International Software Product Line Conference Florence Italy*, volume 1, page 10, 09 2014.

[23] W. Huang, C. He, and Z. Li. A comparison of implementations for aspect-oriented javascript. In *International Conference on Computer Science and Intelligent Communication (CSIC 2015)*, Zhengzhou, China, 2015.

[24] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482, Apr. 2016.

[25] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[26] C. Kastner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232, Kyoto, Japan, Sept. 2007. IEEE.

[27] H. A. Kurdi. Review on aspect oriented programming. *International Journal of Advanced Computer Science and Applications*, 4(9), 2013.

[28] K. Lee and K. C. Kang. Feature Dependency Analysis for Product Line Component Design. In J. Bosch and C. Krueger, editors, *Software Reuse: Methods, Techniques, and Tools*, volume 3107, pages 69–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[29] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 191–202, Porto de Galinhas Brazil, Mar. 2011. ACM.

[30] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for AOP. In *Proceedings of 8th International Conference on Software Reuse, ICSR 2004*, LCNS 3107, Madrid, Spain, 2004. Springer.

[31] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, Oct. 2004.

[32] P. Michalco and V. Vraňič. Prípady použitia a témy v prístupe theme/doc. Master's thesis, FIIT STU, 2009. Diplomová práca.

[33] J. Perdek and V. Vranić. Lightweight aspect-oriented software product lines with automated product derivation. In A. Abelló, P. Vassiliadis, O. Romero, R. Wrembel, F. Bugiotti, J. Gamper, G. Vargas Solar, and E. Zumpano, editors, *New Trends in Database and Information Systems*, pages 499–510, Cham, 2023. Springer Nature Switzerland.

[34] K. Pohl, G. Böckle, and F. Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer-Verlag, Berlin, Heidelberg, 01 2005. ISBN 978-3-540-24372-4.

[35] pure::systems. PLE & code—managing variability in source code, 2020. URL `https://youtu.be/RlUYjWhJFkM`.

[36] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd international conference on Aspect-oriented software development - AOSD '03*, pages 11–20, Boston, Massachusetts, 2003. ACM Press.

[37] A. Rashid, J.-C. Royer, and A. Rummler, editors. *Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way.* Cambridge, 09 2011.

[38] S. She and T. Berger. Formal Semantics of the Kconfig Language, 01 2010. URL `https://www.eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf`.

[39] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 461–470, Waikiki, Honolulu HI USA, 05 2011. ACM.

[40] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the linux kernel a software product line? In *Proceedings SPLC workshop on open source software and product lines*, 01 2007.

[41] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In G. Goos, J. Hartmanis, J. van Leeuwen, and E. Jul, editors, *ECOOP'98 — Object-Oriented Programming*, volume 1445, pages 550–570. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. Series Title: Lecture Notes in Computer Science.

[42] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 47–60, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306348.

[43] J. van Gurp and J. Bosch. Separation of concerns: A case study. Technical report, University of Groningen, 2002.

[44] B. Veer and J. Dallaway. The ecos component writer's guide, 2001. URL `https://ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html`.

[45] V. Vranić and P. Michalco. Are themes and use cases the same? *Information Sciences and Technologies, Bulletin of the ACM Slovakia (Special Section on Early Aspects, AOSD 2010)*, 2:66–71, 01 2010.

[46] H. Washizaki, Y. Nagai, R. Yamamoto, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, and N. Sugimoto. AOJS: aspect-oriented javascript programming framework for web development. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software - ACP4IS '09*, page 31, Charlottesville, Virginia, USA, 2009. ACM Press.

[47] webmaster@os. The l4re microkernel, 2023. URL `https://os.inf.tu-dresden.de/fiasco/`.

[48] B. Zhang, M. Becker, T. Patzke, K. Sierszecki, and J. E. Savolainen. Variability evolution and erosion in industrial product lines: a case study. In T. Kishi, S. Jarzabek, and S. Gnesi, editors, *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, pages 168–177. ACM, 2013.

# A. Publications

## A.1 Articles in International Scientific Journals

**J. Perdek (80%)** and V. Vranić. Fully Automated Software Product Line Evolution With Diverse Artifacts. IEEE Access, 13: 27325-27358, 2025. IEEE. doi: 10.1109/ACCESS.2025.3539868. (SJR Q1 / JCR Q2)

**Cited by**:

> H. Chemingui. Crafting product configuration guidance through process mining support. Business Process Management Journal. 2025 Apr 30.

**J. Perdek (80%)** and V. Vranić. Automated Assessment of Software Product Line Configuration Expressions Through Code Complexity Metrics. Proceedings on Engineering Sciences, 2025. Accepted. (SJR Q3)

## A.2 Articles in Proceedings of International Scientific Conferences

**J. Perdek (80%)** and V. Vranić. Complexity of In-Code Variability: Emergence of Detachable Decorators. In Proceedings of 21st International Conference on Software Reuse, ICSR 2024, LNCS 14614. Limassol, Cyprus. Springer, 2024. doi: 10.1145/3698322.3698357. (Scopus)

**J. Perdek (80%)** and Valentino Vranić. Matrix Based Approach for Structural and Semantic Analysis Supporting Software Product Line Evolution. In Proceedings of 10th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2023. Bratislava, Slovakia. CEUR Workshop Proceedings, 2023. (Scopus)

**Cited by**:

> Y. Kataieva and M. Nemec. Determining software quality using code analysis metrics. 29th International Conference on Information Technology (IT). Zabljak, Montenegro. 2025. pp. 1-4, doi: 10.1109/IT64745.2025.10930266

**J. Perdek (80%)** and Valentino Vranić. Lightweight Aspect-Oriented Software Product Lines with Automated Product Derivation. In New Trends in Databases and Information Systems: ADBIS 2023 ADBIS 2023 Short Papers, Doctoral Consortium and Workshops: AIDMA, DOING, K-Gals, MADEISD, PeRS, CCIS 1850, Modern Approaches in Data Engineering and Information System Design, MADEISD 2023, a workshop at 27th European

Conference on Advances in Databases and Information Systems, ADBIS 2023. Barcelona, Springer, Spain. Springer, 2023. (Scopus)

**Cited by**:

O. Udvardi and J. Lang. Engineering Learning Content Through Question and Answer Pairs. SN Computer Science. 6, 485 2025. https://doi.org/10.1007/s42979-025-04011-3

## A.3 Presentations at International Scientific Conferences

1. Lightweight aspect-oriented software product lines with automated product derivation
   **Place:** Barcelona (UPC), Spain, 2023

2. Matrix based approach for structural and semantic analysis supporting software product line evolution
   **Place:** Bratislava (FIIT), Slovakia, 2023

3. Complexity of in-code variability: Emergence of detachable decorators
   **Place:** Limassol (St. Raphael Resort), Cyprus, 2024

4. Automated Assessment of Software Product Line Configuration Expressions Through Code Complexity Metrics
   **Place:** Oxford (online), United Kingdom/online, 2025

## A.4 Articles in Preparation

**J. Perdek (80%)** and V. Vranić. Aspect-Oriented Software Product Lines with No Aspects in Products. Submitted to International Journal of Computing and Digital Systems, 2025. (SJR Q3)
**J. Perdek** and Valentino Vranić. Enforcement of Variability Comprehension By Preserving Feature Models in Code. To be submitted to IEEE Access. (SJR Q1 / JCR Q2)