Multi-View Editing of Software Product Lines with PEoPL

Mukelabai Mukelabai
Chalmers | University of Gothenburg
Sweden

Benjamin Behringer
htw saar
Germany

Moritz Fey
htw saar
Germany

Jochen Palz
htw saar
Germany

Jacob Krüger
Harz University & University of Magdeburg, Germany

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

ABSTRACT

A software product line is a portfolio of software variants in an application domain. It relies on a platform integrating common and variable features of the variants using variability mechanisms—typically classified into annotative and compositional mechanisms. Annotative mechanisms (e.g., using the C preprocessor) are easy to apply, but annotations clutter source code and feature code is often scattered across the platform, which hinders program comprehension and increases maintenance effort. Compositional mechanisms (e.g., using feature modules) support program comprehension and maintainability by modularizing feature code, but are difficult to adopt. Most importantly, engineers need to choose one mechanism and then stick to it for the whole life cycle of the platform. The PEoPL (Projectional Editing of Product Lines) approach combines the advantages of both kinds of mechanisms. In this paper, we demonstrate the PEoPL IDE, which supports the approach by providing various kinds of editable views, each of which represents the same software product line using annotative or compositional variability mechanisms, or subsets of concrete variants. Software engineers can seamlessly switch these views, or use multiple views side-by-side, based on the current engineering task. A demo video of PEoPL is available at Youtube: https://youtu.be/wByUxSPLoSY

KEYWORDS

Projectional Editing, Product Lines, Annotative, Modular

ACM Reference Format:
Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183499

1 INTRODUCTION

Software-product-line engineering (SPLE) is an approach to implement a portfolio of system variants in an application domain [1]. Instead of engineering variants separately, a software product line (SPL) exploits commonalities and manages variabilities among the variants as features, which are integrated into a common software platform. SPLE provides processes, principles, and mechanisms to effectively engineer such platforms. By selecting the desired features, individual variants can be derived from the platform using an automated and tool-supported process.

To engineer such a platform, developers need to use variability mechanisms—implementation techniques to implement variable software artifacts. Many different variability mechanisms have been proposed [1, 7], typically classified into annotative mechanisms—such as the C preprocessor (CPP) [15] or templates—and into compositional mechanisms—such as feature modules (e.g., AHEAD [3] and FeatureHouse [2]) or deltas [17]. A key difference between these mechanisms is the way they represent features of the SPL. Annotative mechanisms represent features in the source code by wrapping them with variability annotations (e.g., `#ifdef`). While they are relatively easy to apply, annotative mechanisms quickly increase the maintenance effort of the platform [15]. Specifically, they challenge code comprehension, since annotations obscure the structure and the control flows of source code, and since feature code is often scattered across the platform [14, 16]. In contrast, compositional mechanisms represent all of a feature's artifacts in one module, thereby providing a clear code structure that eases code comprehension and decreases maintenance effort. However, decomposing a system into modules is challenging, as it relies on the right decomposition strategy of a system into features, and implementing modules imposes development overhead [1].

All existing SPLE approaches force developers to choose one of these two mechanisms to represent a feature and its software artifacts, and to adhere to this mechanism during evolution and maintenance. While refactorings and combinations have been proposed to switch between annotative and compositional representations [9, 10, 12], they are heavyweight and do not allow developers to quickly switch between different representations of a feature, or even to use representations side-by-side. Previously, we have presented the PEoPL[1] (Projectional Editing of Product Lines) approach [4, 5], which aims at providing developers with the flexibility to exploit benefits of different feature representations on demand. PEoPL relies on separating the internal representation of feature artifacts from the external ones shown to the developer. The latter are currently implemented as seven editable views among which SPL engineers can flexibly switch or which they can even use side-by-side. These views represent the feature artifacts using different annotative and compositional mechanisms, or represent subsets of the variants (hiding unnecessary variability).

[1] http://peopl.de

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183499

In this paper, we demonstrate the PEoPL IDE targeting developers who engineer variant-rich systems, such as SPLs. Based on a running example—an SPL of a simple text editor—we demonstrate the use and advantages of PEoPL. We discuss the engineering challenges PEoPL addresses, briefly introduce its concepts, then exercise four of the seven editable views that engineers can use, and finally discuss case studies and our evaluation.

2 BACKGROUND AND CHALLENGES

To motivate PEoPL, we discuss characteristics and shortcomings of annotative and compositional variability mechanisms.

2.1 Background

Annotative representations. Perhaps the most commonly used annotative mechanism is the CPP, which uses textual variability annotations, such as `#ifdef`, to annotate code of variable features. A similar mechanism can be found in CIDE [11], which relies on graphical annotations (highlighting feature code using background colors). Such annotations have presence conditions (PCs)—Boolean expressions over features—determining the inclusion of respective code by evaluating the PCs of a feature selection.

Compositional representations. Recognizing shortcomings of annotative mechanisms, various compositional mechanisms have been proposed, representing feature code as cohesive units. They are typically called compositional units, feature modules, or delta modules, depending on the concrete variability mechanism. Variant features are defined by creating, in a step-wise fashion, more specific modules of a base module. During product derivation, these modules are composed to derive the desired variant.

In contrast to annotative representations, compositional representations avoid scattered feature code and thereby support comprehension: Developers can inspect one module to understand even highly scattered features. However, compositional mechanisms impose overhead—such as additional code to define a module or implement module interactions—and finding the right decomposition of a system is challenging. As such, they have found less adoption, but their advantages, especially when evolving and maintaining features, call for SPLE approaches that facilitate their adoption.

2.2 Engineering Challenges

Implementing a set of reusable systems as an SPL causes issues related to variability. In particular, PEoPL addresses the following.

First, the complementary variability mechanisms—annotative and compositional—have different pros and cons considering the way they present an SPL's code to the user [1, 9]. To utilize the pros of both mechanisms, it is reasonable to combine or allow an engineer to switch between them—allowing annotated and modular views on the same code. This way, engineers can use the presentation they see most fit for a specific task or compare them in parallel.

Second, only combining the complementary variability mechanisms breaks *uniformity* and, as a consequence, can cause inconsistencies and other problems in the source code [12, 13]. To avoid such problems, despite allowing different presentations of the code to the engineer, PEoPL relies on a uniform internal structure of the source code based on an abstract-syntax-tree (AST). Thus, using different presentations for the engineer does not cause inconsistencies in the code. In addition, the uniform internal structure facilitates extending PEoPL.

Third, *compositional presentations* pose new challenges, such as coarser granularity and missing context information [9, 13]. PEoPL mitigates these, as engineers can integrate annotations into modules to enable fine-grained variability and modules can be blend in or out of the base code. Due to the separation of internal structure and external presentations, PEoPL allows engineers to customize their views on the code to avoid granularity and context issues.

3 THE PEOPL APPROACH AND IDE

To illustrate the external representations of PEoPL, we use the Simple-Text-Editor (STE) SPL shown in Figure 3. In STE, users can choose variants of a text editor based on ten features, including, for example, text highlighting, line wrapping, and specific programming language support.

To provide developers with different representations of an SPL on demand, we introduce the PEoPL IDE—a projectional editor for SPLs. PEoPL is implemented upon the language workbench JetBrains Meta Programming System (MPS) which enables projectional editing. Unlike text editing (a.k.a., parser-based editing), where users modify concrete syntax, i.e., edit characters and a parser builds an AST, in projectional editing user editing gestures directly modify the AST of a program [6].

PEoPL uses projectional editing to enable engineers to choose an external representation—which they interact with—for a feature artifact by separating it from the internal representation of the SPL. Figure 1 illustrates this concept. Internally, the SPL is represented as an AST whose nodes are annotated based on our variability language CoreVar, which is language-independent. Externally, developers are free to choose annotative or compositional representations, or both in parallel for a software artifact. Any editing activity performed by engineers is directly applied on the AST.

PEoPL's CoreVar can be tailored to any language, which allows to even provide more advanced representations as the one shown in Figure 2. As a result, the major benefits of PEoPL are:

- A uniform internal representation to support diverse external representations, and a simple way to plug in new external representations;
- the possibility for developers to switch between external representations of an artifact on demand;
- the ability to edit an artifact using different representations in parallel.

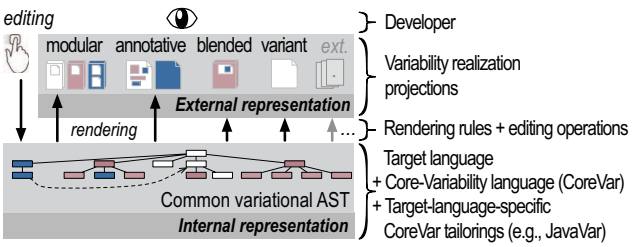

Figure 1: PEoPL separates internal and external variability representations.

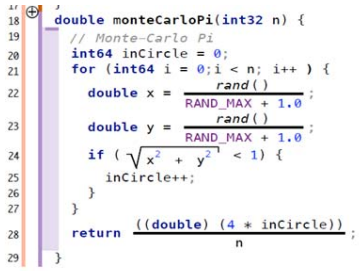

Figure 2: PEoPL's CoreVar language tailored to a mathematical language.

PEoPL supports seven external representations that developers can use when engineering an SPL: Textual annotation projection, visual annotation projection, module projection, blending projections, variant projection, fade-in module projection, and reuse projection. We now focus on four projections and explain them in more detail.

Textual Annotation Projection. As it is the most commonly used variability mechanism, the CPP representation is supported by PEoPL. In Figure 3a we show this projection, in which the finer granularity of annotations can be quicker achieved compared to using modular representations. PEoPL also supports both, disciplined and undisciplined annotations of the source code. Undisciplined annotations are those that do not align with the source code structure: They do not wrap entire classes, methods, or statement blocks. While often considered problematic, they are still used in practice.

Visual Annotation Projection. Since textual annotations can quickly clutter the code and make it hard to comprehend, a visual representation is provided in PEoPL. Similar to CIDE [11], this representation uses coloring to represent parts of the code that implement a particular feature. In Figure 3b, we show the same code as in Figure 3a, but with colors on the left to separate features. For instance, code implementing `HighlightMode` is highlighted purple and code implementing `Statistics` is highlighted green, aiming to improve visibility of features and readability of code.

Variant Projection. Often, engineers want to assess a specific variant (e.g., for a customer) in detail. For this purpose, PEoPL provides a variant projection, where artifacts that do not correspond to the current feature selection are hidden. Figure 3c illustrates a view of the base configuration.

Module Projection. It is often the case that engineers work on source code for one particular feature during a task. For example, if an engineer edits the feature `Statistics`, it is much more convenient and simpler if the corresponding code is in one place (physically or logically). Thus, the engineer does not have to scroll through multiple lines of colored or annotated code. To this end, PEoPL provides the module projection view of features, which we display in Figure 3d. With this projection, developers can quickly locate and edit code related to a feature of interest.

4 CASE STUDIES AND EVALUATION

As case studies, we migrated four annotative and three modular SPLs, and developed one SPL (Jest) from scratch in PEoPL. Table 1 summarizes their characteristics. These SPLs cover different domains, sizes, and original variability mechanisms. In a first evaluation, we analyzed three characteristics of these case studies and compared them to the original variability mechanisms [5]. First, we considered whether PEoPL can *express the same variability* as the two variability mechanisms. This is the case, since we could migrate all SPLs in Table 1 without workarounds. Second, we investigated the *scalability* of creating variants of an SPL. Here, we found that PEoPL scales well for systems up to the size of Berkeley DB—requiring 5.2 seconds on average for generating and writing 2,000 variants to the disk, compared to 18 seconds for the compositional FeatureHouse and 7 seconds for the annotative CIDE. Third, we measured the overhead of a pure compositional mechanism, considering boilerplate code that is required to enable the same variability of the CIDE SPLs with compositional mechanisms. We found that relatively few methods require actual boilerplate code (e.g., 13% in Berkeley DB). However, due to fine granularity and feature interactions within method bodies, almost all of these methods require boilerplate code—due to reliance on external representations instead of changing the internal one.

Table 1: Software product lines available in PEoPL.

SPL	LOC	CLA	F	FM	Source	Domain
Berkeley DB	19k	218	42	83	CIDE	Database
GPL	1k	15	21	26	CIDE	Graph
Java-Chat	0.6k	8	9	9	CIDE	Chat
Jest	19k	144	22	22	-	Web search
Lampiro	45k	140	19	19	CIDE	Instant messaging
Prop4J	2k	6	14	14	FeatureHouse	Propositional formula
STE	1k	9	10	10	DeltaJ	Text editor
Vistex	2k	9	16	16	FeatureHouse	Graph/text editor

LOC: Lines of code | CLA: Classes | F: Features | FM: Feature modules

In a second evaluation, we conducted a user study with seven Master students, in which we analyzed the usability of PEoPL. Despite the small sample size, we could already observe that all participants have been capable of implementing SPLs with PEoPL. The results indicate that our participants prefer visual annotations to textual ones. Considering the different projections, we found that two students solely used visual annotations, but the other five stated that switching and parallelizing views is useful.

5 RELATED WORK

Other development environments for SPLs, such as FeatureIDE [18], enable different implementation techniques. However, only one technique and its representation can be used at a time. To overcome this problem, some approaches aim to combine, integrate, or migrate between annotations and modules [8, 10, 12, 13]. In contrast to PEoPL, none of these allows to use different external representations in parallel—especially as tool support is missing—and can cause several problems due to non-uniform mechanisms.

Projectional editing for SPLs has been proposed before [19]. This led to the development of the mbeddr [20] language family, which also uses MPS to apply CPP concepts on an AST. To our knowledge, this is the only other approach for projectional editing on SPLs but does not allow for multiple external representations.

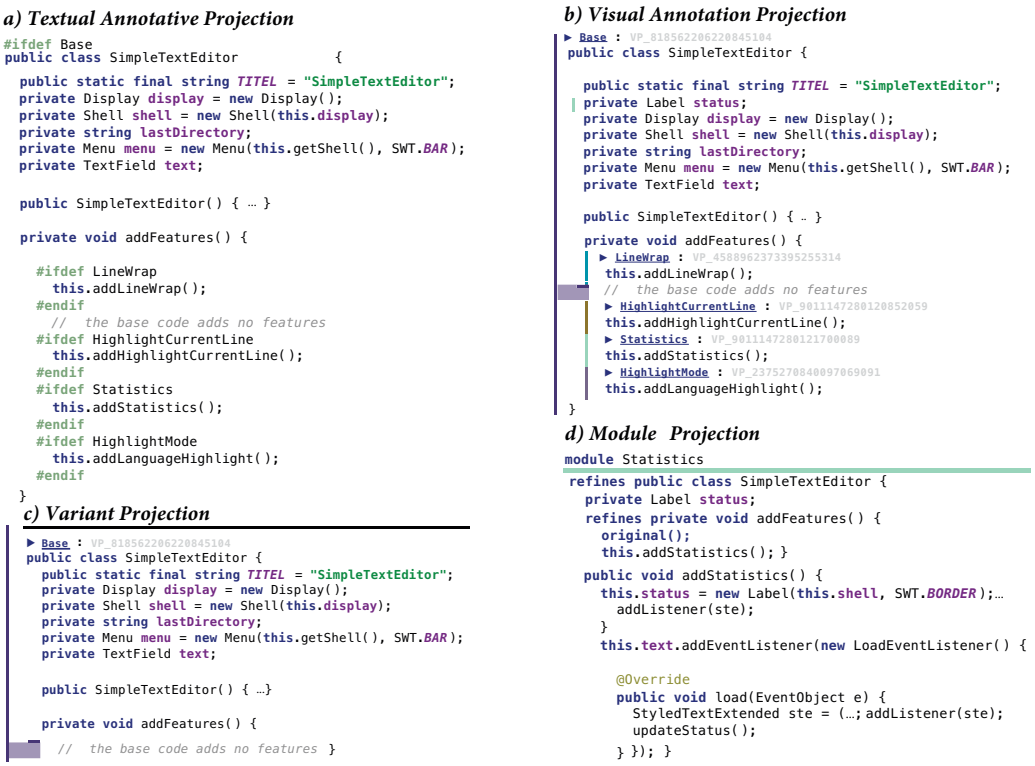

Figure 3: Excerpt of the Simple Text Editor SPL with different external presentations.

6 CONCLUSION

In this paper, we presented the PEoPL IDE for multi-view editing of SPLs. Using a unified internal representation that is separated from external representations, PEoPL combines advantages of annotative and compositional variability mechanisms—allowing to use both in parallel and on-demand for the same software artifact. Currently, we provide seven complementing external representations in PEoPL and a full IDE based on the projectional language workbench MPS.

ACKNOWLEDGMENTS

Supported by the ITEA project REVaMP^2 funded by Vinnova Sweden (2016-02804), by the Swedish Research Council Vetenskapsrådet (257822902), and the German Research Council DFG (3382/2-1).

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer.
[2] Sven Apel, Christian Kästner, and Christian Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 39, 1 (2013), 63–79.
[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (2004), 355–371.
[4] Benjamin Behringer. 2017. *Projectional Editing of Software Product Lines - The PEoPL Approach.* Ph.D. Dissertation.
[5] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE.*
[6] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *FSE.*
[7] Cristina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. *ACM SIGSOFT Software Engineering Notes* 26, 3 (2001), 109–117.
[8] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. 2003. XVCL: XML-Based Variant Configuration Language. In *ICSE.*
[9] Christian Kästner and Sven Apel. 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *McGPLE.*
[10] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Geatures. In *GPCE.*
[11] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code. In *ViSPLE.*
[12] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2018. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience* 48, 3 (2018), 402–427.
[13] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *FOSD.*
[14] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE.*
[15] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor - An Interview Study.. In *ECOOP.*
[16] Leonardo Passos, Jesus Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *MODULARITY.*
[17] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *SPLC.*
[18] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE - An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85.
[19] Markus Völter. 2010. Implementing Feature Variability for Models and Code with Projectional Language Workbenches. In *FOSD.*
[20] Markus Völter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. mbeddr: An Extensible C-Based Programming Language and IDE for Embedded Systems. In *SPLASH.*