Layered Programming: A Language Independent Variability Management Approach

J. De Pauw

University of Antwerp
joey.depauw@student.uantwerpen.be

C. Gomes

University of Antwerp
claudio.gomes@uantwerp.be

H. Vangheluwe

McGill University, University of Antwerp
hv@cs.mcgill.ca

Abstract

Many techniques to implement software product lines exist. Examples are feature-oriented programming, aspect-oriented programming and delta-oriented programming. They are all bound to a specific set of source languages. We propose a way of encoding variability independent of the used language. The goal is to simplify software product line implementation, making it accessible to non-experts. A tool/protocol is used to achieve this goal, much like git is used for version control.

Keywords: software product line, variability management, feature model

1. Introduction

Layered programming is the concept of writing code in layers. A software product line is represented by a common base and a set of layers. A layer can be seen as an overlay or delta to some program that adds, removes or changes functionality. We use the name "layer" to represent a feature refinement. This term is often used in the context of feature-oriented programming.[3]

It is possible to define a layer on top of another layer, providing a hierarchy. Layers can also be independent of each other. They can then be combined with specific semantics, encoded in a feature diagram. A Layer is defined by a reference to its base layer and a set of differences with respect to the base.

Note that a layer is not limited to one language or one file. Every system has multiple representations, like source code, makefiles, documentation and so on. Adding a feature to a program should elaborate each of its representations so they are all consistent.[3, 17]

Though the name "layered programming" suggests a programming paradigm, it is more closely related to the category of tool support. The technique does not aim to replace the need for a good SPL oriented architecture, but rather to simplify the process of implementing, documenting, organizing and managing the architecture.

In this study, we address following topics:

1. formal definition and workflow description
2. use cases for layered programming
3. implementation outline for a tool to support layered programming
4. analysis of the risks involved

This paper is structured as follows. Section 2 contains the related work. In section 3 a more detailed motivation is given. Sections 4 and 5 propose a working example and explain the solution with layered programming respectively. In section 6 more technical aspects and risks are considered. Finally sections 7 and 8 conclude and list future work.

2. Related Work

Birk et. al. investigated SPL practices in the software industry in their paper [4] from 2003. They remark the following about SPL architecture and tool support:

"All products should fit into the provided architecture and benefit from it. Unfortunately, the common architecture's functionality, interfaces, and constraints are usually abstract and complex. Not all the development organization's members or teams will understand them well. Not knowing the SPL architecture's capabilities inevitably leads to the architecture not being fully used."

"Because requirements engineering for SPL can become highly complex, effective tool support is important. Existing tools dont satisfactorily support aspects such as variability management, version management for requirements collections, management of different views on requirements, or dependency modeling and evolution."

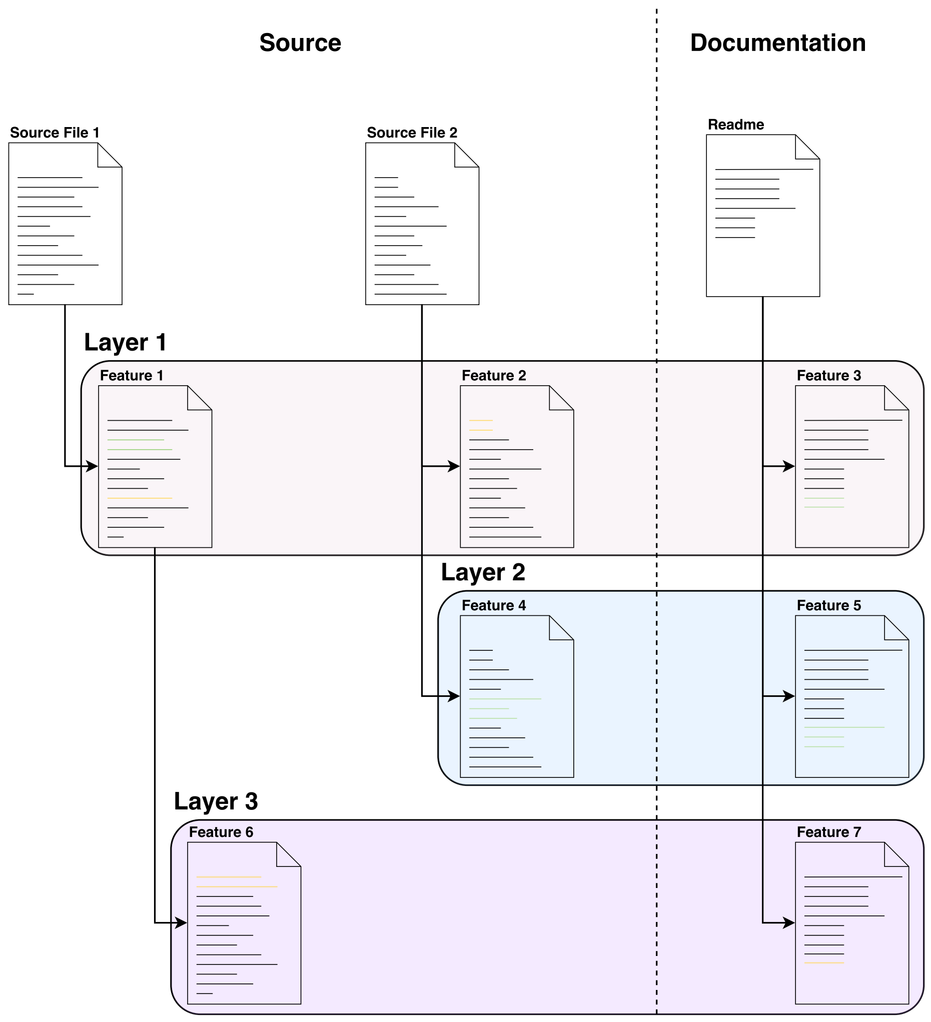

Figure 1: Visual representation of layered programming

Layered programming is a tool-supported technique to assist in SPL development. One of the benefits is that the variability is not hidden away in different files like in most other techniques. Programmers can immediately see all features that affect the piece of code they are working on, preventing code clones and duplicate features.

Since 2003 numerous tools for SPL engineering have been proposed, among which CIDE[6], AHEAD[3], FeatureHouse[1], FeatureIDE[16, 7] and VariantSync[11].

The creation of CIDE was motivated by the problems of both compositional and annotative approaches. It is an Eclipse-based prototype tool for decomposing legacy applications into features that may have a fine granularity[6]. Features are not annotated directly in the code (like preprocessor directives). CIDE manages the feature information and indicates what code belongs to which feature by using different background colors.

The workflow of this tool is closely related to that of layered programming. One of the common advantages is that feature code is still placed where it extends the program, and it is therefore obvious to see how it extends the program, it is simple to understand how a feature is implemented. The main differences are that CIDE uses language specific information (AST representation) to encode features. The CIDE workflow starts with a fully composed application with all features implemented in a single code base, typically a legacy application. It then supports the removal of features to create an artifact. Layered programming implements both bottom up and top down approaches to create software product lines. Artifact are created by adding and combining features, rather than removing them.

AHEAD shows that software can have an elegant, hierarchical mathematical structure that is expressible as nested sets of equations. AHEAD tools are capable of generating Java and non-Java artifacts automatically from nested sets of equations using the Jak-specific tools jampack or mixin.[3, 14] AHEAD uses the Jak language to describe features and to compose them in layers. The technique described in this paper is language independent and does not introduce a new language.

FeatureHouse is a general architecture of software composition supported by a framework and tool chain. FeatureHouse provides facilities for feature composition based on a language-independent model of software artifacts and an automatic plug-in mechanism for the integration of new artifact languages.[1] This tool generalizes many feature-oriented approaches by providing the typical extend/override mechanism on a language-independent model. It is a descendant of Batorys AHEAD program generator.[1, 3]

FeatureIDE is arguably the most popular open-source framework for feature-oriented software development (FOSD). It is based on Eclipse and supports several FOSD implementation techniques such as feature-oriented programming, aspect-oriented programming, delta-oriented programming, and preprocessors.[16] Numerous other tools have been built on top of the FeatureIDE architecture (e.g.: CIDE, VariantSync, BUT4Reuse).

Shaefer et. al present the idea of delta-oriented programming (DOP) and pure DOP in [12, 13].

Delta-oriented programming is a flexible programming language approach. A product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. A product implementation for a particular feature configuration is generated by applying incrementally all delta modules to the core module.[12]

It relates to layered programming in the sense that it allows a programmer to defined deltas (called layers in this paper) with respect to a core module. Deltas are specified syntactically and grouped together by their functionality. The technique described in this paper is purely tool-based and allows the definition of features right in the core module.

Mezini et. al. address the shortcomings of classic object oriented development in [8]:

“Classes as the traditional units of organization of object oriented software have proved to be insufficient to capture entire features of the software in a modular way. As a result, the last decade has seen quite a number of approaches that concentrate on a more appropriate representation of features in the source code”

They explore language specific solutions like aspect oriented programming and feature-oriented approaches in the context of variability management and investigate their shortcomings. Feature-oriented approaches are defined as a class of approaches that concentrate on encapsulating features as increments over an existing base program, together with a mechanism for combining different features on demand. Existing feature-oriented approaches (FOAs) include: GenVoca [2], mixin layers [14, 15], delegation layers [10], and AHEAD [3].

The technique proposed in this paper can be classified as a language independent feature-oriented approach, not to be confused with feature-oriented programming.

In their paper from 2014, Thüm et. al. conclude that feature-oriented software development (FOSD) provides several techniques for the implementation of SPLs. But, each technique comes with advantages and disadvantages, and that there is no consensus on the best technique.[16]

Like other techniques for domain implementation in SPLs, this one too has its advantages and disadvantages. In section 3 it is compared to existing techniques.

3. Motivation

To overcome the increasing demand for tailored software systems, industrial software development often uses clone-and-own to build a new variant by copying and adapting an existing variant. Indeed, this procedure is easy to use and requires less up-front investments. However, with an increasing number of variants, development becomes redundant and the maintenance effort rapidly grows. Hence, at some point, a sufficient number of variants is reached and the migration to a product line is necessary. However, using a product line to develop variants has several downsides. First, product lines have high up-front investments which make the development of few variants unprofitable. Hence, introducing a product line would be a risky task that could not pay off if the number of required variants is unknown at beginning of development.[11]

To overcome this problem, domain implementation techniques should not only support proactive, but also reactive and extractive product line engineering. In reactive product line engineering, only a basic set of products is developed. When new customer requirements arise, the existing product line is evolved. The extractive approach allows turning a set of existing legacy application into a product line. Development starts with the existing products from which the other products of the product line are derived. [13, 5]

Layered programming can be deployed for proactive, reactive and extractive product line engineering. Layers can easily be derived and extracted from existing code bases. Since this is a purely tool-based technique, there is no need to add code. Depending on the quality of the existing architecture, refactoring may not even be required.

One of the hardest tasks in software product line engineer is complexity management. Often abstract and complex architectural designs are conceived to support variability (e.g.: mixins and mixin layers) or new languages are invented (e.g.: DeltaJ, Caesar, AHEAD). Complexity can even arise from extensive use of preprocessor directives (ifdef).

Though each of these techniques have their advantages and disadvantages, we can conclude that they all add complexity to a software product line, making domain implementation a task for SPL experts rather than domain or language experts, which is counter intuitive. With the technique presented in this paper, we aim to decrease the threshold for programmers to create software product lines and simplify the transition form a single system to a family of systems.

TODO Alternative to ifdef (preprocessor directives), AOP, FOP, DOP, ...

TODO Compare to other FOAs

TODO No crosscutting

The most prominent advantages are listed and described:

3.1. Easy to use

TODO no new language to learn, no new structures/architectue

3.2. Semantically clear

TODO features visible at the place where they affect the system.

TODO no intermediate representation

3.3. Robuust

TODO features stay up to date with respect to base

3.4. Timeless

TODO - timeless, works for all languages, even those to be created

4. Working Example

TODO Example from code or stockinfo/pricing example from [8]

5. Workflow

TODO User actions to create example

TODO Refer to tool, editor and feature model

6. Technical

Up until now, we have focused on what the desired result is of layered programming. This section describes how it can be achieved. Three logical components are required to realize the workflow described in section 5:

- a program for extracting/encoding layers and applying them
- an editor to visualize the layers
- a feature model to describe valid combinations of layers

6.1. Program

Two elementary operations are needed to support all the features of layered programming: *extract* and *apply*. Extract takes two files and produces a patch to convert the first file into the second. Apply uses this patch and applies it to a file. These operations closely relate to the *diff* and *patch* algorithms.

Some constraints have to be met on the implementation for *extract* and *apply* operations to ensure a correct result:

- a patch needs to remain applicable under minor, independent changes to its base
- conflicting patches need to be detected and reported, rather than being applied anyways.

The first constraint is clearly needed in the case the base needs to be changed. It is also needed because it has to be possible to apply multiple independent patches consecutively. A degree of fault detection is ensured by the second constraint. Patches are applied in a fuzzy way, based on the context, which is allowed to change. Applying a patch in the wrong place can result in hard to find and hard to fix bugs.

6.1.1. Diff & Patch

We first investigated a possible implementation using the Linux diff and patch commands. The diff command generates a patch file that can be used to turn one file into the other with the patch command. Note that the -u flag was used to make sure the unified format was used with a number of context lines.

We quickly discovered that the line difference calculated by the diff command is not fine grained enough. Layers need to be allowed to make independent changes on the same line like for example adding an argument to a function.

Another option is to use a character based difference algorithm. Myers et. al. proposed a performant algorithm for this in [9]. It is implemented in Google's diff-patch-match library.

TODO Refrase above

TODO line diff - word diff - char diff

TODO use full semantics of source language: tree diff

TODO use heuristics (class, def, function, types, parantheses, ...)

6.1.2. Encoding

TODO Think of good, user friendly encoding

TODO Should support multiple files

TODO Investigate interaction with version management (git)

6.2. Editor

TODO Define features of editor

TODO Suggest implementation as plugin for jetbrains (intelij) or atom

6.3. Feature Model

TODO Layers organized in feature model

TODO Combination semantics encoded in FM

TODO FM in editor

6.4. Risks

TODO Is it robuust under changes to base

TODO What should happen with conflicting features?

TODO failed/wrong patch not detected leads to broken artifact.

TODO (system tests are needed to detect this)

TODO interaction with version control. How to resolve merge conflict?

TODO No crosscutting support (can be added maybe?)

TODO How to deal with optional layer interaction?

```
e.g.: B -> F1, B -> F2
what if F2 needs to change added code of F1
solutions:
- fixed with crosscutting support?
- new layer F3 from F1 and F2 (F1, F2 -> F3)?
- require F3 if F1 and F2 present in feature model?
- Allow F2 to immediately encode optional changes to F1 (put F3 part in F2).
```

7. Conclusion

TODO write

8. Future Work

TODO write

[1] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[2] Don Batory and Sean O'malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[4] Andreas Birk, Gerald Heller, Isabel John, Klaus Schmid, Thomas von der Maßen, and Klaus Muller. Product line engineering, the state of the practice. *IEEE software*, 20(6):52–60, 2003.

[5] P Clements and CW Krueger. Being proactive pays off/eliminating the adoption barrier. point-counterpoint article in. *IEEE Software*, 2002.

[6] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 311–320. IEEE, 2008.

[7] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. Featureide: Empowering third-party developers. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pages 42–45. ACM, 2017.

[8] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 127–136. ACM, 2004.

[9] Eugene W Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[10] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP*, volume 2, pages 89–110. Springer, 2002.

[11] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing software variants with variantsync. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 329–332. ACM, 2016.

[12] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, pages 77–91, 2010.

[13] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 49–56. ACM, 2010.

[14] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. *ECOOP98Object-Oriented Programming*, pages 550–570, 1998.

[15] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[16] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[17] Wikipedia. Feature-oriented programming — wikipedia, the free encyclopedia, 2017. [Online; accessed 4-December-2017].