Integrating Approaches for Feature Implementation

Benjamin Behringer
University of Luxembourg, Luxembourg
& htw saar, Germany
benjamin.behringer@htwsaar.de

ABSTRACT

Compositional and annotative approaches are two competing yet complementary candidates for implementing feature-oriented software product lines. While the former provides real modularity, the latter excels concerning expressiveness. To combine the respective advantages of compositional and annotative approaches, we aim at unifying their underlying representations by leveraging the snippet system instead of directories and files. In addition, to exploit this unification, we propose different editable views.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

Software product lines, features, modularity, virtual separation of concerns, snippets

1. RESEARCH PROBLEM

In feature-oriented software development, features are employed to define and communicate variable and common artifacts of a software product line [3]. From an end-user's feature selection, a product variant can be generated automatically. To make features explicit in source code, two common approaches exist: *compositional* and *annotative* ones [13].

In compositional approaches such as FEATUREHOUSE [4] and AHEAD [6], feature implementations are separated into feature modules, which can be composed to produce a product variant. This physical separation provides real modularity, meaning locality and cohesion [15]. However, compositional approaches lack expressiveness [13] and thus require developers, who intend to implement fine-grained extensions, to either introduce code clones or hook methods.

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2666605

Annotative approaches such as the *C preprocessor* and *coloring* [13][5] support fine-grained extensions inherently [13]. Features are assigned virtually by annotating base code fragments. To generate a variant, annotated fragments can be excluded conditionally. However, the support of fine-grained extensions comes at the expense of real modularity [15].

In summary, there is a "trade-off between expressiveness and modularity mechanisms in current solutions" [15]. Consequently, a practical approach that puts the expressiveness of fine-grained extensions and modularity benefits into concert is currently missing, yet demanded.

2. RELATED WORK

Like others [5][12][14][24][25], we propose to integrate compositional and annotative approaches to combine their respective advantages. Batory et al. [5] as well as Walkingshaw and Erwig [24] provide valuable theoretical research in this respect. Kästner et al. discuss an integration [12], and put it into practice by presenting a model for refactoring from each representation into the other [14]. However, annotative approaches are more expressive and thus refactoring renders hook methods inevitable [14].

With their tool CIDE, Kästner et al. propose to emulate modularity in annotative approaches by hiding the code and even files associated with a feature on demand [13]. To avoid ordering issues, visual markers appear in the source code for hidden features. Moreover, CIDE only supports adding feature details, not removing or replacing them [25][5]. We build upon the ideas introduced with CIDE, but omit visual markers and support the overriding of feature details [7].

Walkingshaw and Ostermann build upon these ideas as well, and propose a formal model of editable views on variational software [25]. Thereby, a generic *abstract syntax tree* (AST), which may contain choices, represents a program. This AST is used to generate different (simplified) documents, which in turn can be edited, parsed and committed back into the generic AST. As a result, their approach allows adding, removing and replacing feature details. We represent the source code as a single common graph that supports alternative paths (i.e., choices). Yet, instead of parsing and committing the changes made to a document, modifications are carried out directly on the underlying graph.

The Decal tool of Janzen and Devolder is not targeted at software product lines, but provides two alternative views on a common code base [11]. The code, which is stored in a database, can be extracted (i.e., restructured) into a representation of the program that is decomposed into either classes or modules that crosscut classes. To commit modifi-

cations, a parsing step is required. Thus, in contrast to our approach, modifications are not made directly to the underlying representation. Moreover, instead of restructuring the source code, we hide feature details and build different interface mechanisms on top (i.e., at the tool level).

Altogether, *homogeneous crosscutting concerns* [3] are not made explicit in the aforementioned approaches. We fill this gap by providing appropriate interface mechanisms (e.g., by emerging a modular interface in the IDE on demand). Moreover, our approach is closely related to projectional editors, where a developer directly modifies the AST instead of a temporary representation (i.e., concrete syntax) [23]. Yet, in contrast to classical tool-centric approaches such as IPE [19] and MPS[1], we pursue a document-centric approach on top of the *snippet system* [17][18].

In prior work, we picked up the mathematical foundation for an integration of compositional and annotative approaches provided by Batory et al. [5] and leveraged it in the field of risk assessment [8]. Due to the lack of tool support, we could merely emulate our ideas using CIDE. Thus, we came up with the ideas presented in this proposal. Moreover, we provided a first step towards an integration by means of a vision paper [7].

3. RESEARCH METHODS AND SKETCH OF THE APPROACH

We plan to put into practice the integration of compositional and annotative approaches by providing a unified underlying representation of the source code. To guide future work, we define our research questions as follows.

RQ1. *How is it possible to unify the underlying representations of compositional and annotative approaches?*

RQ2. *Does a unified representation help to overcome the current trade-off between expressiveness and modularity?*

With regard to RQ1, we plan to leverage the snippet system instead of statically organizing implementations by means of directories and files. The snippet system proposes a novel operating system environment that involves a fine-granular document model mapping documents onto graphs of so-called *snippets*. An inherent advantage thereof is the ability to share snippets across a document (or documents). Consequently, the snippet system supports the creation of new product variants manually by means of sharing snippets between documents. To narrow RQ1 and RQ2, we formulate the following corresponding hypotheses.

H1. *Representing compositional and annotative approaches as colored snippet graphs enables unification.*

H2. *A unified representation provides means to implement views and interface mechanisms that help to overcome the current trade-off between expressiveness and modularity.*

To answer RQ1, we need to adapt the snippet system. This adaption is necessary to support the automatic generation of variants from a given feature selection. In particular, we propose to explicitly assign features and feature interactions to individual snippets using coloring. Moreover, we enable the labeling of edges with Boolean formulas to support alternative paths in the graph. As a result, we propose a snippet-based model of an integrated representation.

[1]http://jetbrains.com/mps

To answer RQ2, aside from compositional and annotative views on the unified representation, additional interface mechanisms are needed. These interfaces should support the integration of the advantages of a compositional view into an annotative one and vice versa. We are mainly interested in interface mechanisms that help to deal with fine-grained extensions, information hiding, quantification and weaving. In particular, we plan to provide a compositional view that supports information hiding and fine-grained extensions as well as an annotative view that supports quantification and weaving. Aspect-oriented research concerned with interfaces [2][10][22] and the transformation of annotated code into aspects [9][20][1] is related to our field of study and provides rough guidance [15]. Moreover, an *emergent interface* strategy offers interesting opportunities [21]. For instance, using an annotative view, a modular interface can be *emerged* on demand by leveraging the snippet graph.

3.1 Overview

Figure 1 provides an abstract overview of our ideas. In the graph, we use different graphical elements for nodes to indicate the different semantics of snippets. In detail, we define that a snippet represents either an organizational unit or in case of source code an AST element. To outline the illustrated concepts, we proceed from bottom to top [7].

Source code representation. Essentially, we follow the lead of Kästner et al. and represent the source code as colored ASTs [13][16]. In addition, as we use a graph, alternative paths can be introduced in the AST and its elements can be reused (cf. Fig. 1a). These mechanisms are valuable for (1) the avoidance of traversing wrappers in terms of a color mismatch; (2) the local detection of ordering issues; (3) the realization of homogeneous crosscutting concerns [7].

Organizational units. We propose a set of organizational units that can be seen as a replacement for classical directories and files. A *compilation unit* is a snippet that represents the root of a (shared and) colored AST. Moreover, as we leverage graphs, it is possible to aggregate all extensions introduced by a feature using a compilation unit (cf. Fig. 1b). *Composition units* in turn aggregate subsets of compilation units (i.e., to enable locality and cohesion, a composition unit aggregates all virtual compilation units it refines or introduces). *Collections* enable compositional and annotative views. While the collection for a compositional view aggregates all composition units, the annotative counterpart collects all compilation units of the product line. Of course, similar to directories, additional layers of aggregation are possible.

Views on the source code. Organizational units are employed to browse through and modify the structure of the

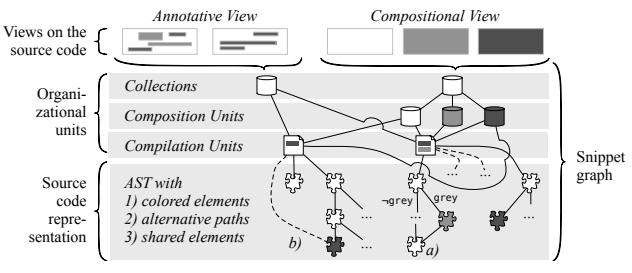

Figure 1: Abstract overview of our approach

product line. In terms of source code, we use the (shared and) colored AST of compilation units to synthesize editable annotative and compositional views (i.e., documents). Developers can choose the one view that suits best for a given task in order to edit the source code. Moreover, to support different levels of complexity, the annotative view is user definable. Modifications are carried out directly on the underlying snippet graph, not on the respective document. Thus, parsing is not required. From a compositional perspective, base-code documents can be virtually duplicated on demand and assigned to a composition unit for the implementation of a feature. Thereby, the implementation details of all other features are hidden. Moreover, we plan to provide interfaces that help to deal with fine-grained extensions as well as homogeneous ones. For instance, an interface that reveals the semantics of an annotated code fragment (i.e., whether it is a heterogeneous or homogeneous crosscutting concern) can be emerged in the IDE.

3.2 Realization

To implement the source code representation and the organizational units, we adapt the *snippet model* [17] in order to support coloring and alternative paths. To synthesize documents automatically for views and emergent interfaces, we adapt the *authoring tools* [17][18] of the snippet system. Thereby, for every open (and shared) entity, an instance of a managing tool, which assures that modifications to the snippet graph are made persistent, is running. Moreover, modifications are revealed in all other views automatically.

4. PROGRESS

The snippet model specifies a well-defined structure for snippet graphs [17]. *Element snippets* represent structural elements that are organized as an ordered tree. A *document snippet* in turn constitutes the root of such a tree (or trees), once reuse capabilities are demanded. As a logical consequence, an element snippet can link a document snippet to exploit these reuse capabilities.

We have provided an initial step towards answering RQ1 in prior work [7]. There, we have introduced (1) the idea of a snippet-based approach to separate features, (2) the organizational units, and (3) the variability-aware source code representation. Altogether, we use document snippets to represent organizational units and link them using element snippets. Moreover, we employ element snippets for representing AST elements and their order, and leverage document snippets to reuse these AST elements in turn. The combination of both snippet types enables emulating edges that are labeled with a Boolean formula. Finally, coloring is selected to assign features to snippets [13][16][5].

In Figure 2, we illustrate how to leverage snippets on the level of source code representation (cf. Fig. 1). On the left, there are two feature implementations from a compositional point of view: *Base* and *BarCheck*. *Base* introduces a method `foo` that is refined by *BarCheck*. In particular, *BarCheck* introduces an `IfStatement` checking the parameter `bar` of the method `foo`. Blue-colored dashed lines represent the feature implementations introduced by *BarCheck*.

On the right hand side of Figure 2, we illustrate an excerpt of the snippet graph that reflects the implementations. AST elements are represented by element snippets, for instance the method declaration of `foo` and the `IfStatement`. To share AST elements and provide alternative paths in the

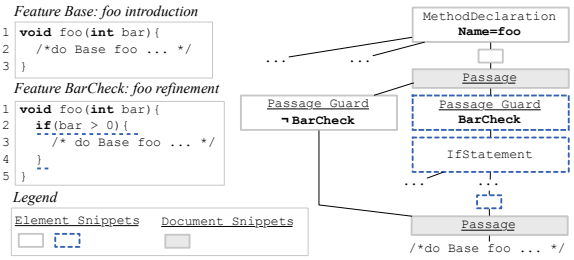

Figure 2: Implementing features using snippets

AST, we introduce the *passage* concept [7]. A passage is a document snippet and thus a potential point of variability. In particular, the bottom passage in Figure 2 enables sharing the base code of `foo`, while the top passage is the entry point for an alternative path in the AST. Just as coloring, *passage guards* control access to their following AST elements during traversal (i.e., each cursor position corresponds to a position in the snippet graph). Thereby, a passage guard can be associated with a Boolean formula, while each Boolean variable corresponds to a feature (e.g., of a feature model).

To test H1, we show that the snippet model and our proposed concepts are generally suitable to unify the representation of compositional and annotative approaches. Therefore, we match the mathematical foundation for an integration provided by Batory et al. [5] with our ideas of employing the snippet system. Batory et al. generalize the standard ideas of variation points (VPs), fragments and modules by providing a broader definition than the one used in tools like CIDE. They point out that (1) a VP may appear more than once across a document (or documents), and if a fragment fills one instance of the VP, all other instances are filled with it as well; and (2) a set of fragments can be assigned to a VP, while at most one fragment can fill the VP at a time. We fulfill these requirements with our snippet-based approach [7]. A document snippet represents a VP that can be linked by multiple element snippets. This linkage represents the appearance of the VP across a single or a set of document(s). Consequently, clause (1) is fulfilled. In addition, a passage guard represents the conditional assignment of a fragment to the VP. As a result, only one fragment can fill the VP at a time, and thus clause (2) is complied as well.

For a final answer to RQ1, we need to further develop our concepts. For instance, we currently enable the sharing of entire compilation units, AST elements and their subtrees, and allow implementing fine-grained extensions using a compositional view [7]. In addition, we need concepts to realize quantification and weaving. We expect our current concepts to provide useful means to achieve this goal.

5. EVALUATION

As mentioned before, we test H1 by picking up the mathematical foundation for an integration of compositional and annotative approaches presented by Batory et al. [5], and show how the theories map onto our snippet-based model.

In addition, to stress our hypotheses and evaluate our ideas, we plan to perform different case studies. First, we intend to develop feature-oriented software product lines from scratch using our snippet-based approach. Second, we plan to refactor existing case studies of compositional [4] and annotative research [13] into our snippet environment and vice

versa. Thereby, we plan to provide a fully automated refactoring for FEATUREHOUSE and CIDE as well as a semi-automated approach for projects, where the C preprocessor is employed. Aside from variability, the C preprocessor is used for *include guards*, *macros* and even *constants*. Thus, clear guidelines must be devised in a dedicated research question to support such a semi-automated refactoring.

As a result, we will provide means to test H2 and the feasibility of our novel interface mechanisms. Thereby, we plan to empirically evaluate our proposed views and interfaces with regard to development effort, comprehensibility, maintainability and other aspects.

6. EXPECTED CONTRIBUTIONS

With our research, we lay ground for a snippet-based integration of compositional and annotative approaches. We unify the underlying representations of the source code by means of a single common snippet graph. Moreover, we provide views and interface mechanisms that support persistent modifications that are directly carried out on the graph. In particular, we expect to make the following contributions to overcome the current "trade-off between expressiveness and modularity" [15].

(1) Our approach supports compositional and annotative views on real documents. Thereby, we focus on the advantages of real modularity. Scattered fragments do not need to be searched. Moreover, implementations are not organized by statical directories and files, but by means of flexible snippet graphs that enable to share information across a document (or documents).

(2) Using passages, we support alternative paths and the sharing of AST elements. Consequently, we avoid the traversal of subtrees in case of a color mismatch and support the overriding of feature details. Moreover, passages enable a straightforward detection of conflicting fine-grained extensions across feature modules. For instance, ordering issues can be detected automatically and handled locally [7].

(3) We provide a snippet-based compositional view that supports information hiding. Thus, private variables and methods of the base code are hidden for feature modules. Moreover, fine-grained extensions can be implemented without requiring code clones or hook methods.

(4) We offer a snippet-based annotative view that supports quantification and weaving. Consequently, instead of requiring developers to switch to a compositional view, we plan to emerge an interface that reveals the semantics of an annotated code fragment on demand.

(5) Our ideas to leverage the snippet system reveal interesting opportunities for related disciplines such as code clone research. For instance, sharing and thus reusing AST elements can help to avoid code clones. Thus, our proposed concepts might be useful to support or even improve development paradigms such as *copy, paste & adapt programming*.

7. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *AOSD*, 2009.

[2] J. Aldrich. Open Modules: Modular Reasoning About Advice. In *ECOOP*, 2005.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013.

[4] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.*, 2013.

[5] D. Batory, P. Höfner, B. Möller, and A. Zelend. Features, modularity, and variation points. In *FOSD*, 2013.

[6] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 2004.

[7] B. Behringer, L. Kirsch, and S. Rothkugel. Separating features using colored snippet graphs. In *FOSD*, 2014.

[8] B. Behringer, M. Lehser, and S. Rothkugel. Towards feature-oriented fault tree analysis. In *QUORS*, 2014.

[9] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *AOSD*, 2007.

[10] W. Chae and M. Blume. Language support for feature-oriented product line engineering. In *FOSD*, 2009.

[11] D. Janzen and K. De Volder. Programming with Crosscutting Effective Views. In *ECOOP*, 2004.

[12] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *McGPLE*, 2008.

[13] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, 2008.

[14] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *GPCE*, 2009.

[15] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *SPLC*, 2011.

[16] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *TOOLS EUROPE*, 2009.

[17] L. Kirsch, J. Botev, and S. Rothkugel. Snippets and Component-Based Authoring Tools for Reusing and Connecting Documents. *JDIM*, 2012.

[18] L. Kirsch, J. Botev, and S. Rothkugel. The Snippet Platform Architecture: Dynamic and Interactive Compound Documents. *ICIMT*, 2013.

[19] R. Medina-Mora and P. H. Feiler. An Incremental Programming Environment. *IEEE Trans. Software Eng.*, 1981.

[20] A. Reynolds, M. E. Fiuczynski, and R. Grimm. On the feasibility of an AOSD approach to Linux kernel extensions. In *ACP4IS*, 2008.

[21] M. Ribeiro, P. Borba, and C. Kastner. Feature Maintenance with Emergent Interfaces. In *ICSE*, 2014.

[22] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM TOSEM*, 2010.

[23] M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *SLE*, 2014.

[24] E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. In *GPCE*, 2013.

[25] E. Walkingshaw and K. Ostermann. Projectional Editing of Variational Software. In *GPCE*, 2014.