Towards Feature Modularization

Márcio Ribeiro Paulo Borba

Informatics Center, Federal University of Pernambuco, 50740-540, Recife – PE – Brazil

{mmr3, phmb}@cin.ufpe.br

Abstract

Virtual Separation of Concerns was introduced as a way to reduce drawbacks of implementing product line variability with preprocessors. Developers can focus on certain features and hide others of no interest. However, features eventually share elements, which might break feature modularity, since modifications in a feature result in problems for another. In this thesis we propose the concept of emergent feature modularization. The idea consists of establishing contracts among features to prevent the developer from breaking other features when performing a maintenance task.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Design

Keywords Product Lines, Modularity, Preprocessors

1. Introduction

In a Software Product Line (SPL), features are often implemented using mechanisms like preprocessors [2], so that directives such as `#ifdef` and `#endif` encompass code associated with features. Despite their widespread use, several drawbacks are known, including no support for separation of concerns. Virtual Separation of Concerns (VSoC) [2] allows developers to hide feature code not relevant to the current task, being important to reduce some of the preprocessors drawbacks. The idea is to provide developers a way to focus on a feature without being distracted by other ones.

Although VSoC is helpful to visualize a feature individually, it does not modularize features to the extent of supporting independent feature maintenance and development [3], since developers know nothing about what is hidden. In fact, when maintaining a feature, a developer might introduce errors into the hidden features, since these features eventually share elements (variables and methods) with the feature being maintained. For instance, the new value of a variable might be correct to the maintained feature, but incorrect to another that uses this variable. Thus, we have a problem due to the lack of feature modularization: the modification of a feature leads to errors in another. And this problem is worse since this error would only be noticed when running the product built with the problematic feature.

This thesis proposes the concept of *Emergent Feature Modularization* [4], which consists of establishing contracts among feature implementations. We call our approach emergent because the components and interfaces here are neither predefined nor have a rigid structure. Instead, they emerge on demand to give support for specific feature development or maintenance tasks. Notice that we also achieve the hiding benefits towards feature comprehensibility. However, while still hiding completely the feature code, our emergent interfaces abstract its details. At the same time, they provide valuable information to maintain a feature and keep other features and their possible combinations safe. Our intent is to provide enough information to prevent developers of breaking other features, even when they are working on parallel.

Our hypothesis is that, by using the emergent interfaces, developers achieve modularity and, consequently, make fewer mistakes during SPL maintenance, improving their productivity. In particular, our research questions are the following. **Q1**: Do emergent interfaces provide better support during feature maintenance?; **Q2**: Do emergent interfaces allow developers to analyze less code?

2. Emergent Feature Modularization

The top of Figure 1 shows two features of Mobile Media[1]: `Music` and the `Copy` optional feature (implemented with preprocessors). We do not provide the `Copy` feature code on purpose to simulate VSoC, so that the developer is not concerned about other features like, for this example, `Copy`. To some extent, hiding features is worthwhile to the feature comprehensibility benefit, since it may help developers to comprehend a feature individually. Despite this advantage, VSoC does not provide enough support for feature modularization, which also means modifying features separately [3].

[1] `http://mobilemedia.cvs.sourceforge.net/`

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

Because there is no information about the hidden code, when maintaining the `Music` feature, problems may occur in `Copy`. So, the independent *changeability* benefit is not achieved. For example, since the `screen` variable is used only at the `MMController` constructor, a developer may decide to change `MMController(screen)` to `MMController(new MMScreen(..))` and delete the `screen` declaration. Since the `Copy` feature uses `screen`, an error will occur when a developer eventually compiles the product with the problematic feature combination (with `Copy`).

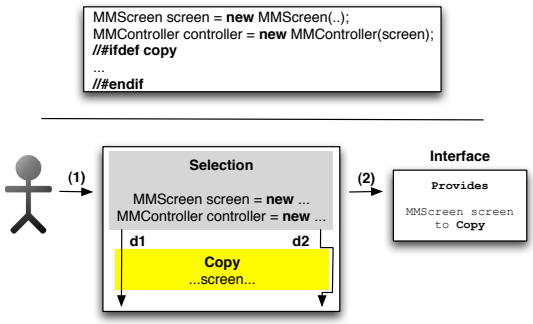

Figure 1. Copy feature hidden / Emergent Interface.

In this context, sharing information about two or more features may be a confusing point for two developers, so that achieving the *parallel development* is difficult. This happens because there is no "*mutual agreement between the creator and accessor*" [5]. Since this contract does not exist, developers of a feature might actually break another one.

To solve these problems, we propose the concept of emergent feature modularization, which consists of establishing, according to a given development task, interfaces among features. This is based on an uncommon way to think about components and interfaces: they are not predefined, nor have a rigid structure, but are computed on demand, to give support for feature development. For example, in a maintenance, the feature code to be changed is a component, named *Selection*. The backward/forward paths of the code surrounding it are components too. Paths consider the different feature combinations by the feature model. They are named *dataflows*, since data is exchanged among features. Interfaces capture data dependencies between these components, and give support to maintaining *Selection* without having to understand the details of code associated to the *dataflows*.

Thus, before changing the `Music` feature, developers select the code to be maintained. In this case, since `Copy` is optional, two *dataflows* are considered according to the feature model: **d1:** `Music` $\wedge$ `Copy` and **d2:** `Music` $\wedge$ ($\neg$ `Copy`). They are illustrated through arrows on the bottom of Figure 1. After the selection, interfaces emerge to basically show data dependencies between components. The *dataflows* are used to catch dependencies between the selection and code of other features. Figure 1 shows an emergent interface, stating that the *Selection* component provides `screen` to the `Copy` feature. This interface allows us to change *Selection* abstracting details of surrounding features (which are still hidden). At the same time, they provide information to the *Selection* developer, so that he might avoid implementations that cause problems to other features. Now, when looking at the interface, he would think twice before continuing the refactoring. Now we present the ongoing work and some results.

More evidences to our problem. We are trying to collect semantic errors caused by the lack of feature modularization. Also, the problem addressed here gets worse depending on the number of features within methods. For this reason, we are computing for some C and Java product lines metrics like the number of `#ifdefs` per method; and the number of variables declared in a feature and used in another one.

Tool. We are building a tool (which is based on *Colored IDE* [2]) to compute emergent interfaces.

Evaluation. For **Q1**, we should collect real scenarios of SPL maintenance, like adding, removing, and changing features. By using these scenarios, we intend to conduct an experiment with students to evaluate if our proposal allows developers to commit fewer mistakes. For **Q2**, since our approach provides information about what is hidden, we count the lines of code of the hidden feature and of our interfaces. Our interfaces should be smaller. Otherwise, it seems to be easier for the developer to analyze the hidden code directly.

How do we go beyond? We still have the VSoC benefits since hidden feature details are abstracted. At the same time we provide summarized information to maintain a feature and keep the hidden ones safe. Therefore, emergent interfaces help developers to change a feature without breaking others. Thus, we may achieve not only the comprehensibility benefit, but also the independent changeability. Some works check for type errors of all SPL variants [1]. Our intent is to make developers aware about other features before initiating the maintenance, avoiding errors that would be only caught afterwards by these checking-based works. Finally, we are also concerned with system behavior, rather than only with static type information. For example, interfaces may state that a feature needs a particular value for a variable.

References

[1] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd ASE'08*, pages 258–267. IEEE Computer Society, September 2008.

[2] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th ICSE'08*, pages 311–320, New York, NY, USA, 2008. ACM.

[3] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.

[4] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. In *Proceedings of the Onward! 2010*, New York, NY, USA, 2010. ACM. To appear.

[5] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, 1973. ISSN 0362-1340.