Visual Support for Understanding Product Lines

Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt
University of Magdeburg
{janet.feigenspan, ckaestne, mfrisch, dachselt}@ovgu.de

Sven Apel
University of Passau
apel@uni-passau.de

***Abstract*—The C preprocessor is often used in practice to implement variability in software product lines. Using #ifdef statements provokes problems such as obfuscated source code, yet they will still be used in practice at least in the medium-term future. With CIDE, we demonstrate a tool to improve understanding and maintaining code that contains #ifdef statements by visualizing them with colors and providing different views on the code.**

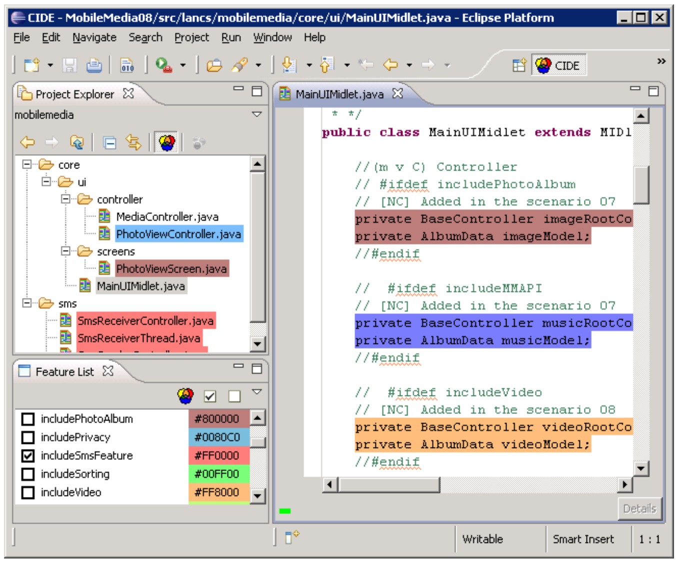
I. Introduction

Software product line (SPL) engineering is an efficient technique to create variable software. Instead of implementing each software product from scratch, program variants are generated from a set of features. A *feature* is a user-visible characteristic of a software system [1] and is modeled and implemented according to requirements of a domain. The source code of those features constitute the SPL, from which different software products, or *variants*, can be generated.

SPLs promise several benefits, including reduced cost and time to market, efficient mass customization, improved quality, and better maintenance and evolution [1]. However, these benefits come at a price of a more complex implementation; instead of a single program, an SPL developer implements multiple (potentially millions of variants) at the same time.

For implementing SPLs, conditional compilation, for example using ifdef directives of the C preprocessor, is used often in practice, despite the well-known problems, such as source code scattering and obfuscation [2]. From HP's product line of printer firmware [3] to the Linux kernel [4], there are many examples of SPLs implemented with conditional compilation. Although academics rather recommend modular implementation mechanisms, such as components or aspects, industrial adoption is slow. Especially when large amounts of legacy code are involved, there is no way around preprocessors in the medium-term future. Hence, we propose a tool to support the use of preprocessors.

We currently explore different approaches to support understanding of SPLs implemented with conditional compilation, by different forms of visualization and other tool support. In our tool CIDE, we display code fragments framed by #ifdef statements, by highlighting them with a background color (one color per feature). Colors clearly differ from the source code (compared to #ifdef statements) and that humans can recognize and process them considerably faster than text. This allows developers to get a quick overview of source code and helps to navigate through source code and locate bugs that are associated with certain features. In addition to



```
 * */
public class MainUIMidlet extends MIDl

    //(m v C) Controller
    // #ifdef includePhotoAlbum
    // [NC] Added in the scenario 07
    private BaseController imageRootCo
    private AlbumData imageModel;
    //#endif

    //  #ifdef includeMMAPI
    // [NC] Added in the scenario 07
    private BaseController musicRootCo
    private AlbumData musicModel;
    //#endif

    //  #ifdef includeVideo
    // [NC] Added in the scenario 08
    private BaseController videoRootCo
    private AlbumData videoModel;
    //#endif
```

Feature List

☐	includePhotoAlbum	#800000
☐	includePrivacy	#0080C0
☑	includeSmsFeature	#FF0000
☐	includeSorting	#00FF00
☐	includeVideo	#FF8000

Fig. 1. Screenshot of the project MobileMedia in CIDE. The feature list (bottom left) shows all features in the project and allows to assign colors. The project explorer (top left) currently shows only files that contain code of the feature "includeSmsFeature", as selected in the feature list (8 out of 51 files). The source code editor also highlights code fragments that belong to a feature with an according background color.

colors, CIDE provides views on the source code to emulate modularity. Although the implementation of a feature is still scattered, at tool level, we represent it similar to a modularized implementation. Initial results indicate that visualizations and tool support as in CIDE can improve code comprehension and developer productivity [5].

In the demonstration, we show and discuss different visualizations and illustrate them in our prototype on existing SPLs.

CIDE was presented in prior work with focus on a discussion of granularity, correctness, and scaling [6], [7]. Here, we focus on program comprehension, especially how CIDE can support a developer to understand (unfamiliar) source code containing #ifdef statements.

II. CIDE

CIDE is an Eclipse plug-in to support SPL development based on conditional compilation. It is released (including further documentation) as open source at http://fosd.de/cide.

A. Visualization

CIDE provides a specialized editor as shown in Figure 1, in which annotations are represented by background colors. Colors are mapped to features, so for example, all code fragments that are included only when feature "includeSmsFeature" is selected are shown with the same background color.

When the entire content of a file is framed with conditional compilation constructs, the entire file is highlighted with the same color in Eclipse's resource browser. If all files are colored, so is the entire directory. This is useful especially for large SPLs, in which sometimes complete packages implement a feature. The developer can recognize the association of a directory or file with a feature in the project explorer without having to open the according file.

Obviously there are scaling issues when using distinct background colors for typical SPLs with several hundred features. Mapping all features to different colors does not scale, because humans cannot distinguish several hundred colors without direct comparison. In CIDE, we simply allow a developer to assign more than one feature to the same color and provide tool tips, which show the underlying feature(s) of a source code fragment. Alternatively, we can assign shades of gray to all feature code and let the developer manually chose colors for features that are relevant for the task at hand. We currently explore different directions; due to space restrictions, we skip a detailed discussion here.

B. Views

To support a developer to hide code fragments that are not relevant for the task at hand, CIDE provides views on the file system and views on file content. A view shows only code fragments that belong to a specific feature or feature combination (which the developer can select in a dialog). When activated, only files that contain code of a selected feature are shown in the resource tree (left in Figure 1). When opening a file, again only code fragments that belong to the feature are shown, which is implemented similar to code folding. Additionally, the view includes some context information; for example, in the case only a single statement is framed, the surrounding class and method name are provided as context information. The context information is similar to context provided in modularized implementations in form of interfaces or pointcuts.

Additionally to views on a feature or feature combination, CIDE also provides a preview function for a specific variant. For a valid feature selection, all included files and code fragments are shown, similar to the code of the generated variant. This allows a preview and detailed analysis, for example, for debugging code that occurs only in the interaction of two features. For more details see [7].

Views are implemented at editor level, without actually modifying source code. Developers can quickly change between views or expand them. Views are editable, so developers can modify code directly inside views (to avoid ambiguities, we show markers for hidden code).

III. Benefits

Using background colors to represent code framed by #ifdef statements can speed up program comprehension. In an experiment, we found a speed up for up to 43 % for some tasks, compared to conventional #ifdef statements [5]. We selected the existing SPL MobileMedia [8], which was implemented with #ifdef statements and created a second version in which variability was represented by colors. We found that when subjects should locate feature code, they were faster when they worked with colors. However, when subjects should fix bugs, colors had at best no effect on the response time. In a follow-up experiment, we additionally found that, when given the opportunity to switch between both representations, subjects use this frequently. We conclude from both experiments, that colors help programmers to work with unfamiliar source code. However, we also found that colors have to be chosen with care, such that programmers are not distracted by them (which happened in one case in our experiment).

So far, we did not measure how views improve program comprehension. Nevertheless, Atkins et al. [9] have found an $40\ \%$ increase in productivity in their tool Version Editor, which provides similar views (views on variants, completely hiding variability from developers). Our tool is more flexibile and allows different kinds of views, but in general we expect a similar improvement.

IV. Summary and Future Work

Currently, we explore different visualization approaches and forms of tool support to improve understanding of software product lines. In parallel, we explore different color palettes, zoomable interfaces, seesoft-style visualizations, actual remodularization instead of views at editor level, and many more. Our general research goal is to support developers who have to deal with conditional compilation.

Acknowledgment

Apel's work is supported in part by DFG project #AP 206/2-1. Feigenspan's work is supported by BMBF project 01IM08003C (ViERforES).

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practice and Patterns*. Addison Wesley, 2001.

[2] J. Favre, "Understanding-In-The-Large," in *Proc. Int'l Workshop on Program Comprehension*. IEEE, 1997, p. 29.

[3] T. Pearse and P. Oman, "Experiences Developing and Maintaining Software in a Multi-Platform Environment," in *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 1997, pp. 270–277.

[4] J. Sincero *et al.*, "Is The Linux Kernel a Software Product Line?" in *Proc. SPLC Workshop on Open Source Software and Product Lines*, 2007.

[5] J. Feigenspan, "Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study," Master's thesis, University of Magdeburg, 2009.

[6] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 311–320.

[7] C. Kästner, S. Trujillo, and S. Apel, "Visualizing Software Product Line Variabilities in Source Code," in *Proc. SPLC Workshop on Visualization in Software Product Line Engineering*. Lero, 2008, pp. 303–312.

[8] E. Figueiredo *et al.*, "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 261–270.

[9] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus, "Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 625–637, 2002.