A Tool for Improving Maintainability of Preprocessor-based Product Lines

Márcio Ribeiro[1,3], **Társis Tolêdo**[1], **Paulo Borba**[1], **Claus Brabrand**[2]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE), Recife, Brasil

[2]IT University of Copenhagen (ITU), Copenhangen, Denmark

[3]Instituto de Computação – Universidade Federal de Alagoas (UFAL), Maceió, Brasil

{mmr3, twt, phmb}@cin.ufpe.br, brabrand@itu.dk

Abstract. *Virtual Separation of Concerns (VSoC) reduces drawbacks when implementing preprocessor-based product lines. Developers can focus on certain features and hide others of no interest. But features eventually share elements between them, which might break feature modularization, since modifications in a feature can result in problems for another. Emergent interfaces can help in the sense they capture dependencies between the feature being maintained and the other ones. Developers then become aware of feature dependencies and are likely to avoid wrong feature implementations. In this paper, we present a tool that implements the emergent concept. We capture feature dependencies by using feature-sensitive data-flow analysis capable of analyzing all variants simultaneously: there is no need to analyze each one separately.*

1. Introduction

A Software Product Line (SPL) is a family of software-intensive systems developed from reusable assets. These systems share a common set of features that satisfy the specific needs of a particular market segment [Clements and Northrop 2002]. In this context, features are the semantic units by which we can differentiate programs in a SPL [Trujillo et al. 2006]. Features are often implemented using preprocessors [Kästner et al. 2008]. Conditional compilation directives such as `#ifdef` and `#endif` encompass code associated with features, mixing common, optional and even alternative behavior in the same code asset.

Despite their widespread use, preprocessors have some drawbacks, including no support for separation of concerns [Spencer and Collyer 1992]. Virtual Separation of Concerns (VSoC) [Kästner et al. 2008] allows developers to hide feature code not relevant to the current task, reducing some of the preprocessors drawbacks. The idea is to provide developers a way to focus on a feature without being distracted by others.

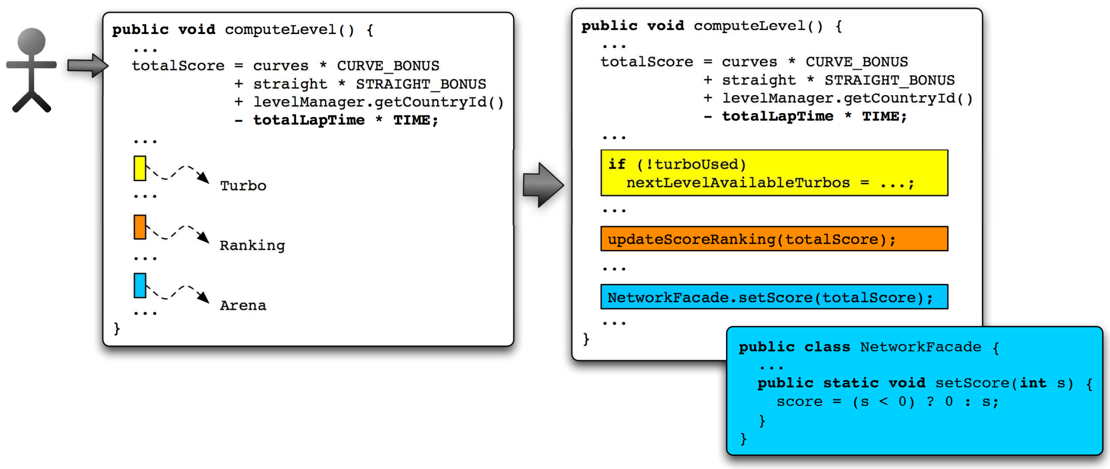
However, VSoC is not enough to provide feature modularization, which aims at achieving independent feature comprehensibility, changeability, and development [Parnas 1972]. In fact, by visualizing and trying to maintain a feature individually, a developer might introduce errors into the other features due to dependencies, since these features possibly share elements—such as variables and methods—with the maintained feature. Thus, we face the lack of feature modularization: because of feature dependencies, the new value of a variable, for example, might be correct to the maintained feature, but incorrect to another one that uses this variable. In fact, these feature dependencies are quite common in practice [Ribeiro et al. 2011].

To minimize these problems, we proposed the concept of *emergent interfaces* [Ribeiro et al. 2010]. The idea consists of capturing dependencies between the feature a programmer is maintaining and the others. These interfaces emerge and provide information about other features that might be affected by maintenance tasks. Developers then become aware of the dependencies and, consequently, might avoid the maintainability problems. Developers still have the VSoC benefits. Emergent interfaces complement VSoC in that in addition to hiding feature code, they provide dependency information.

In this paper, we present a tool that implements the emergent feature modularization concept. Our tool is based on CIDE [Kästner et al. 2008], an implementation for the VSoC concept. Like CIDE, our emergent tool is an Eclipse plug-in. In addition, we use the SOOT [Vallée-Rai et al. 1999] framework to execute data-flow analysis and then capture feature dependencies. Since we consider SPLs, we extend SOOT to implement feature-sensitive data-flow analyses.

2. Motivating Example

The scenario we illustrate here is based on the *best lap*[1] SPL. *Best lap* is a racing game where the player tries to achieve the best time in one lap and qualify for the pole position. In this game, there is a method responsible for computing the game score, as illustrated in Figure 1. The method contains small rectangles, representing hidden features that the developer is not concerned with and thus not seeing. Note that there are no `#ifdef` statements. Instead, the VSoC approach relies on tools that use background colors to represent features, which helps on not polluting the code with preprocessors [Kästner et al. 2008].



```
public void computeLevel() {
  ...
  totalScore = curves * CURVE_BONUS
             + straight * STRAIGHT_BONUS
             + levelManager.getCountryId()
             - totalLapTime * TIME;
  ...
  ...
  ...
  ...
}
```

```
public void computeLevel() {
  ...
  totalScore = curves * CURVE_BONUS
             + straight * STRAIGHT_BONUS
             + levelManager.getCountryId()
             - totalLapTime * TIME;
  ...
  if (!turboUsed)
    nextLevelAvailableTurbos = ...;
  ...
  updateScoreRanking(totalScore);
  ...
  NetworkFacade.setScore(totalScore);
  ...
}
```

```
public class NetworkFacade {
  ...
  public static void setScore(int s) {
    score = (s < 0) ? 0 : s;
  }
}
```

Figure 1. Maintenance correctly accomplished only for products without *arena*.

The hidden features are the following: *turbo*, *ranking*, and *arena*. They are optional and responsible respectively for: increasing the car speed; computing the score ranking; and publishing the score in the network. The method contains a variable responsible for storing the player's total score (`totalScore`).

[1]Best lap is a commercial product developed by Meantime Mobile Creations.

Now, suppose the developer were to implement the following new requirement in the (mandatory) *score* feature: let the game score be not only positive, but also negative. Also, suppose that the developer is using VSoC, so that there are hidden features throughout the code. The developer might well judge that they are not important for the current task. To accomplish the task, he localizes the *maintenance point* (the `totalScore` assignment) and changes its value (see the bold line in Figure 1). Building a product with the *arena* feature enabled and running it may make the developer incorrectly assume that everything is correct, since the negative total score correctly appears after the race. However, when publishing the score on the network, he notices that the negative score is in fact stored as zero (see the right side of Figure 1). Consequently, the maintenance was only correctly accomplished for products without *arena*.

Because there are hidden features, the developer might be unaware that another feature he is not maintaining uses `totalScore` and thus also needs to be changed accordingly to correctly complete the maintenance task. In fact, the impact on other features leads to two kinds of problems. The first one is **late error detection**, since we can only detect errors when we eventually happen to build and execute a product with the problematic feature combination (here, any product with *arena*). Second, developers face **difficult navigation** throughout the code. Searching for uses of `totalScore` might increase developer effort. Depending on the number of hidden features, the developer needs to consider many locations to make sure the modification did not impact other features. Further, it is possible that some—or even all—features might not need to be considered if they did not use the variables that were modified. In our example, the developer is likely to analyze 3 features. However, since only *ranking* and *arena* uses `totalScore`, he would analyze the *turbo* feature unnecessarily, increasing maintenance effort.

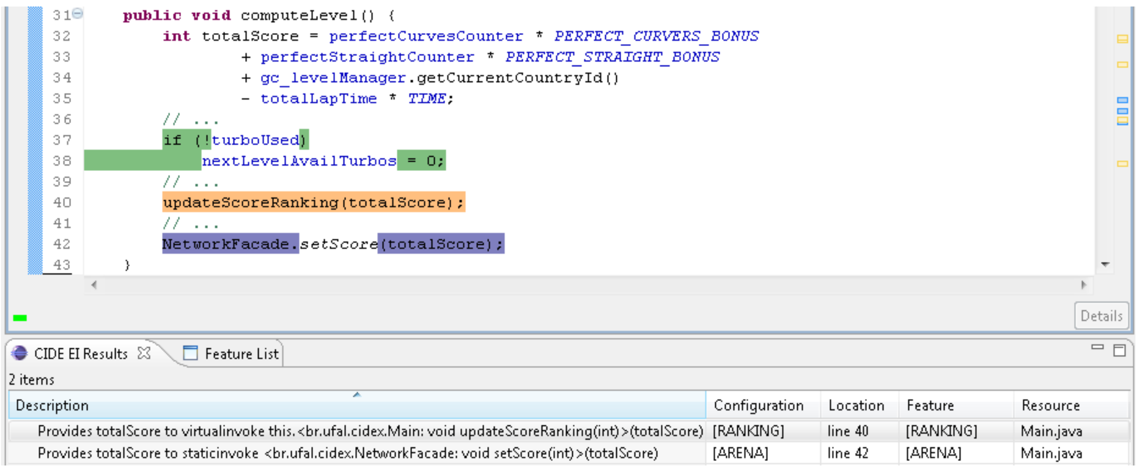
3. A Tool for Emergent Interfaces

Now, we present how emergent interfaces deal with the aforementioned problems. Then, we describe our tool that implements the emergent concept in terms of architecture (Section 3.1) and main functionalities (Section 3.2).

Previously, we presented the Emergent Interfaces [Ribeiro et al. 2010] approach intended to help developers avoid the problems related to feature dependencies. The idea consists of determining, on demand, interfaces for feature implementations. The interfaces are neither predefined nor have a rigid structure. Instead, they **emerge** to provide information to the developer on feature dependencies, so he can avoid introducing problems to other features. To do so, emergent interfaces capture dependencies between the feature we are maintaining and the others. In other words, when maintaining a feature, interfaces emerge to give information about other features we might impact with our maintenance. Notice that emergent interfaces rely on feature code already annotated.

To better illustrate these ideas, consider the scenario of Section 2, where the developer is supposed to change the `totalScore` value. The first step when using our emergent approach consists of selecting the maintenance point. The developer is responsible for such a selection which in this case is the `totalScore` assignment. Then, we perform code analysis based on data-flow analysis to capture the dependencies between the feature we are maintaining and the other ones. Finally, the interface emerges.

To support developers, we extend the Colored IDE (CIDE) [Kästner et al. 2008], a tool that enables feature annotations by using colors and implements the VSoC approach. We name our tool *CIDE EI*[2] (CIDE + Emergent Interfaces). Figure 2 presents the emergent interface in an Eclipse view. It states that the maintenance may impact products containing *ranking* and *arena*. In other words, we *provide* the actual `totalScore` value to both features. The developer is now aware of the dependencies. Reading the interface is important, since the emerged information alerts the developer that he should also analyze those features (see lines 40 and 42 and the "Location" column in the CIDE EI view). When investigating, he is likely to discover that he also needs to modify *arena*, and thus avoid the **late error detection** problem.



```
31    public void computeLevel() {
32        int totalScore = perfectCurvesCounter * PERFECT_CURVERS_BONUS
33                + perfectStraightCounter * PERFECT_STRAIGHT_BONUS
34                + gc_levelManager.getCurrentCountryId()
35                - totalLapTime * TIME;
36        // ...
37        if (!turboUsed)
38            nextLevelAvailTurbos = 0;
39        // ...
40        updateScoreRanking(totalScore);
41        // ...
42        NetworkFacade.setScore(totalScore);
43    }
```

2 items

Description	Configuration	Location	Feature	Resource
Provides totalScore to virtualinvoke this.<br.ufal.cidex.Main: void updateScoreRanking(int)>(totalScore)	[RANKING]	line 40	[RANKING]	Main.java
Provides totalScore to staticinvoke <br.ufal.cidex.NetworkFacade: void setScore(int)>(totalScore)	[ARENA]	line 42	[ARENA]	Main.java

Figure 2. CIDE EI view: code locations and configurations possibly impacted.

Emergent interfaces assist the **difficult navigation** problem since they indicate precisely the product configurations the developer needs to analyze. To achieve this, emergent interfaces rely on feature-sensitive data-flow analysis [Ribeiro et al. 2010]. Thus, our interfaces focus on the configurations we indeed might impact, avoiding developers from the task of analyzing unnecessary features. When clicking on any interface element, the tool points directly to the line described in the "Location" column, being helpful specially in complex code that contains many features.

3.1. Architecture

Figure 3 depicts the architecture our tool.

As mentioned, our tool is based on CIDE, which is an Eclipse plug-in that extends the IDE editor to display annotated and colored source code. Colors are associated with features. Internally, CIDE instruments the nodes of the AST with feature information. The information from the instrumented AST is retrieved to feed SOOT; a Java optimization framework for analyzing and transforming source and bytecode. To analyze code, SOOT transforms it into an intermediary representation, called *Jimple*.

Now, we present the two components we implemented. The *Feature Sensitive* plug-in is responsible for extracting feature information from CIDE and associating it with the Jimple representation of the source code (note the arrows pointing from *Feature Sensitive* to CIDE and SOOT). In SOOT, the data-flow information is stored in ob-

[2]Available at `http://www.cin.ufpe.br/~mmr3/cbsoft2011`

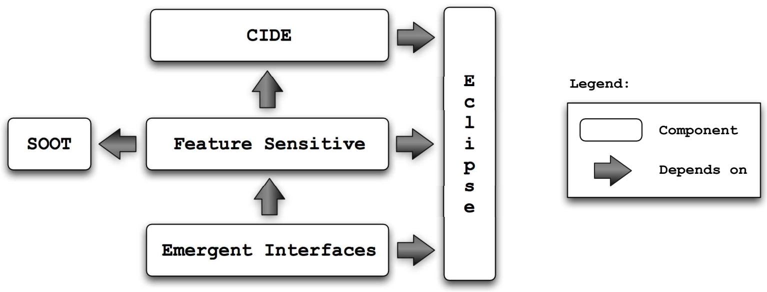


Figure 3. Architecture of CIDE EI (CIDE + Emergent Interfaces).

jects that implements the `FlowSet` interface. We created our own implementation of `FlowSet`—`LiftedFlowSet`—that contains a set of `FlowSet` objects; one for each product configuration. This is important to indicate precisely the product configurations we might reach during our maintenance task.

After computing data-flow analysis for each product configuration, we need to generate the emergent interfaces. The *Emergent Interfaces* plug-in is responsible for this task. First, it calls the *Feature Sensitive* plug-in to obtain data-flow information for each possible configuration within the method being maintained. Then, it computes the emergent interface by crossing the obtained information with the *maintenance point*; finally, it displays the interface to the developer by using an Eclipse view.

The *Feature Sensitive* plug-in implements a more general idea: data-flow analysis taking features into account. The *Emergent Interfaces* plug-in is just one application of such a data-flow analysis to compute interfaces. So, we separated them; note that only the later depends on the former. In fact, the *Feature Sensitive* is capable of executing feature-sensitive analyses in all methods of an entire Eclipse project, not only individual methods as we do in the emergent context.

3.2. Main Functionalities

In summary, our tool provides the following functionalities:

- Computes emergent interfaces that helps us to focus on the features (or feature combinations) we might indeed impact during the maintenance task; when clicking on any element of the interface, the tool points directly to the impacted feature;
- Computes feature dependencies based on feature-sensitive data-flow analyses used to analyze all method variants simultaneously: there is no need to analyze each one separately. Notice that when not using feature-sensitive data-flow analyses, the tool might point out false positives. For example, we might select a variable in feature *A* and the tool indicates variable usage in feature *B*. Since they can be mutually exclusive due to a feature model constraint, the tool points out a false positive, which means that the dependency does not exist;
- Exports feature dependencies graph for a given maintenance point to a DOT[3] file. We intend to create a view based on this graph as well.

[3]`http://www.graphviz.org/content/dot-language`

4. Related Work

Besides CIDE, we relate our tool to the following. Mylyn [Kersten and Murphy 2006] is a task-focused approach to reduce information overload through information hiding, so that only assets (like packages, classes and methods) relevant to a current task are visible. This information is filtered by using a task context that is created during a programming activity. This way, tasks are monitored by Mylyn aiming at storing information about what developers are doing to complete the task. Developers can select a task and Mylyn provides only the assets related to it, improving productivity. Like Mylyn, our approach also needs a selection. The developer selects the snippet to maintain it, whereas when using Mylyn developers select tasks. Our interfaces and the task context of Mylyn emerge during maintenance. We also provide information reduction, since we show only elements shared with other features in the interface.

Conceptual Module [Baniassad and Murphy 1998] is an approach to support developers on maintenance tasks. Developers can set lines of code to be part of a conceptual module and use queries to capture other lines that should be part of it and to compute dependencies among other conceptual modules. Our tool also capture dependencies, but goes further, since it takes feature into consideration. Both approaches abstract details from developers so that they concentrate on relationships among features or conceptual modules rather than on code of no interest, which is important for comprehensibility.

5. Concluding Remarks

This paper presented a tool for computing emergent interfaces in SPLs. The tool helps us to focus on the features (or feature combinations) we might impact with our maintenance. Our tool is based on CIDE to annotate feature code and SOOT to compute data-flow analysis aiming at capturing feature dependencies. To avoid building all possible method variants, we extended SOOT to compute feature-sensitive data-flow analysis.

The tool consists mainly of two Eclipse plug-ins: one implements the feature-sensitive data-flow analysis and the other relies on the first one to compute emergent interfaces. We split our implementation into two different plug-ins so that new applications may use our feature-sensitive implementation of data-flow analysis.

6. Acknowledgments

We would like to thank CNPq, a Brazilian research funding agency, and National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work. Also, we thank SPG[4] members for feedback and fruitful discussions about this paper.

References

Baniassad, E. L. A. and Murphy, G. C. (1998). Conceptual module querying for software reengineering. In *Proceedings of ICSE'98*, pages 64–73. IEEE Computer Society.

Clements, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[4]`http://www.cin.ufpe.br/spg`

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in Software Product Lines. In *Proceedings of ICSE'08*, pages 311–320. ACM.

Kersten, M. and Murphy, G. C. (2006). Using task context to improve programmer productivity. In *Proceedings of FSE'06*, pages 1–11. ACM.

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

Ribeiro, M., Pacheco, H., Teixeira, L., and Borba, P. (2010). Emergent Feature Modularization. In *Onward! 2010, affiliated with SPLASH'10*, pages 11–18. ACM.

Ribeiro, M., Queiroz, F., Borba, P., Tolêdo, T., Brabrand, C., and Soares, S. (2011). On the Impact of Feature Dependencies when Maintaining Preprocessor-based Software Product Lines. In *Proceedings of GPCE'11*. ACM. To appear.

Spencer, H. and Collyer, G. (1992). #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer 1992*, pages 185–198.

Trujillo, S., Batory, D., and Diaz, O. (2006). Feature refactoring a multi-representation program into a product line. In *Proceedings of GPCE'06*, pages 191–200, New York, NY, USA. ACM.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot - a java bytecode optimization framework. In *Proceedings of CASCON'99*. IBM Press.