Modularizing Variabilities with CaesarJ Collaboration Interfaces

Carlos Eduardo Pontual   Rodrigo Bonifácio   Henrique Rebêlo
Márcio Ribeiro   Paulo Borba

Informatics Center, Federal University of Pernambuco Recife, Brazil
{ceplc, rba2, hemr, mmr3, phmb}@cin.ufpe.br

1.   Introduction

Software product lines (SPLs) aim at reducing the time to marketing of applications in a common domain [5]. To achieve that, products are generated by means of *weaving* common behavior, shared by all members of a SPL, with variant behavior that, in fact, implements the variability of each SPL member.

Although several techniques have been used to implement variability in SPLs, an extended notion of interfaces for decoupling common and variant behavior during development is still a challenge. For instance, annotative style [3] for SPL development does not provide a clear separation between common and variant assets—thus, it is not possible, using such a style, to develop both concerns in an independent way.

Besides that, even compositional approaches, such as *aspect-oriented* [4] and *feature-oriented* [2] programming (FOP), which separate common and variant behavior, do not enable the parallel development of common and variant features. This occurs because, in order to modularize a variant behavior as an aspect, developers have to be aware of the details about how the SPL common behavior was implemented. Additionally, to the best of our knowledge, current implementations of FOP do not provide means to specify proper interfaces between common and variant assets. Actually, here we claim for a particular notion of interface, where we should be able to:

1. Clearly state the obligations of different teams, enabling the parallel development of common and variant code.

2. Specify design rules using language constructs, in such a way they could be statically checked by a compiler.

In this paper we investigate the use of Collaboration Interfaces (CI), a particular type of interface supported by Cae-

sarJ [1], to modularize a recurrent and challenging type of variability, where there exists a mutual dependency between the common and variant behavior. We proceed our investigation by assessing different alternatives for implementing one feature present in a Tetris Game Product Line (Section 2). The results we obtained show that, although we had improved modularity using a CI, open issues remain in investigation.

2.   Motivating Example

Consider a simple SPL example of Tetris[1] games. On this SPL, among other variants, we can generate products (i) with normal difficulty, which shows only the next piece that will appear, and (ii) with easy difficulty, showing the next two pieces, as illustrated in Figure 1.
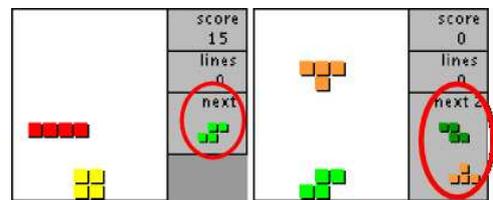

Figure 1. Tetris game. One piece (left), two pieces (right)

In order to implement these variations, two versions of the following methods of `NextPiece` class must be created: (a) `paintBox`, which is called by other methods of `NextPiece` to paint the box that shows the next piece and (b) `updatePiece`, which is called by other classes of the program (e.g., main class) to define what would be the next piece. It is important to note that, when painting the box contents, the `paintBox` method have to access non-variant members of `NextPiece`. A possible implementation of such a variation is by using the Template Method design pattern, which is described in Listing 1.

However, on this solution we have a tangling of design and implementation (problem labeled P1.1). The variation part (abstract methods) can only be implemented after the implementation of the base code. It is not possible to split

---

[1] JSE version of the JME game present on
http://kiang.org/jordan/software/tetrismidlet/

the development between the teams, one responsible for the base part and the other responsible for the variation part. An enhancement to this solution would be the specification of an interface in top of the `NextPiece` class, defining the signature of all the four methods. Although separating the design from the implementation (solving P1.1), this solution brings two problems: (P2.1) it is not possible to explicitly specify in the interface which methods are from the base part and which are from the variation part; and (P2.2) the variation code can not be compiled independently of the base code, because variations extends the class used for the base implementation (see Figure 2(a)).

(a) Template method design.   (b) CaesarJ mixin design.

Figure 2. Template Method and Mixin designs.

Listing 1. Template Method	**Listing 2.** CaesarJ CI
| ```
abstract class NextPiece {
  void paint() { ...
    paintBox(); ... }
  void drawPiece() { ... }
  abstract void updatePiece();
  abstract void paintBox();
}
class Var1 extends NextPiece {
  void setupPiece() { ... }
  void paintBox() { ... }
}
//Simmilar for Var2
``` | ```
cclass NextPieceCI {

  cclass NextPiece {
    void paint();
    void drawPiece();
    //Variation
    void updatePiece();
    //Variation
    void paintBox();
  }

}
``` |

A possible interface to mitigate these problems using CaesarJ is shown in Listing 2[2]. On this solution, we define a design-rule (collaboration interface) named `NextPieceCI`, which defines that the caesar class (`cclass`) `NextPiece` must exists and that this class must have at least the four methods described. With this interface defined, we can split the development between both teams. In that way, each team provides a partial implementation of the `NextPiece` `cclass`, and them both implementations are composed using mixins composition. For instance, the team responsible for the implementation of the base part provides a partial implementation of the `NextPiece cclass` with the methods `paint` and `drawPiece`, while the team responsible for the variations provides two partial implementations, each one with different versions of the methods `paintBox` and `updatePiece`. The selection of which variation will be used together with the base implementation of `NextPiece` is made during the mixin composition, as illustrated on Figure 2(b).

Note that we only have the `cclass NextPiece` on the interface because our variation problem only affect this class. However, we can specify multiple `cclasses` in our interface, depending on the context of our problem.

3. Discussion and Final Remarks

The interface introduced on Listing 2 enables both teams to implement and compile their code independently of each other. Base and variation only depend on the interface now,

contrasting with the Template Method solution, where variations depended on the base implementation (see Figure 2[3]).

Despite improving parallel development (solving P2.2), this solution still has some drawbacks. For instance, problem P2.1 remains open, it is not possible to explicitly specify on the interface which are the roles of each team. We put the "//Variation" comment before the methods of the variation facet, but the compiler is not able to statically check if both teams are implementing the methods assigned to them. The compiler can only verify if all the methods described on the interface are present on the mixin composition, no error will be given if *paintBox*, a method of the base, is implemented on the variation class.

We are currently working on some extensions to the CI concept in order to solve this and other problems that were not covered on this paper. Also, since there is no difference between classes and aspects in CaesarJ, we are trying to use the CIs to write Design-Rules that enables a more modular design between classes and aspects, with focus on the parallel development.

Acknowledgments

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *LNCS*, 3880:135, 2006.

[2] D. Batory. Feature-oriented programming and the ahead tool suite. In *ICSE '04*, pages 702–703, 2004.

[3] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE '08*, pages 311–320. ACM, 2008.

[4] G. Kiczales et al. Aspect-oriented programming. In *ECOOP'97*, pages 220–242, 1997.

[5] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005.

---

[2] Consider all four methods and the `cclass`, as the `NextPiece`, declared as abstract. The keyword was omitted due to lack of space.

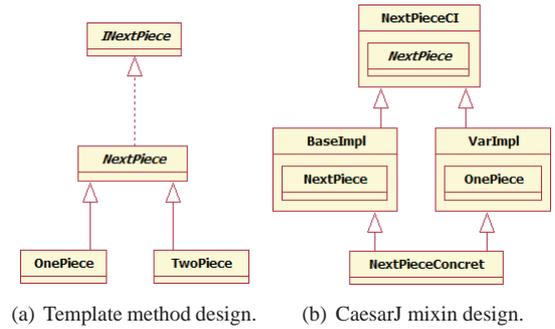[3] Due to the lack of space only one variation was shown in figure 2(b).