On-Demand Integration of Product Lines: A Study of Reuse and Stability

Alessandro Gurgel, Francisco Dantas, Alessandro Garcia
OPUS Research Group – Software Engineering Lab, Department of Informatics, PUC – RJ- Brazil
{agurgel, fneto, afgarcia}@inf.puc-rio.br

ABSTRACT
The integration of multiple SPLs is increasingly becoming a trend to enable on-demand derivation of new products and accelerate their time-to-market. Integration of SPLs often implies the reuse of a previously-implemented feature across other SPLs. The reuse of a SPL feature is only viable if the underlying programming mechanisms enable its smooth composition within the code of other SPLs. If the required modifications are significant, the design of the target SPLs are likely to be destabilized. This paper presents an exploratory study on the integration of three product lines from the board game domain. We investigate how aspect-oriented and feature-oriented programming impact on the reuse and stability of those product lines.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – software product lines; D.2.8 [General]: Metrics – product metrics

General Terms
Languages, Measurement, Design.

Keywords
Software Product Line, Product Line Integration, Stability, Reuse.

1. INTRODUCTION
Software product lines (SPLs) are increasingly being built from other product lines within the same domain [1]. Many reasons can demand the integration of independently-developed SPLs [1][2]. For instance, the reuse of previously-implemented features across those SPLs enables the derivation of additional unique products. The integration of a SPL feature into other SPLs is supported by the underlying composition mechanisms of the programming language. However, if such mechanisms require significant change effort to realize a feature integration, there is not much payoff for the reuse investment being made. In fact, if the amount and scope of modifications are both significant, it is likely the stability of the individual SPL designs will be compromised in short term. A SPL design is stable if the extent of the changes required in its integration process is insignificant [1].

To integrate features from different SPLs, many modifications may be required in the code. The integration of a feature into other product lines may not be a trivial task even when its code is confined to a single module [2]. The reason is that each feature code was not structured with the other SPL features in mind [2]. Many researchers [4][17][6][7] have investigated how advanced programming techniques, such as aspect-oriented programming (AOP) [8] and feature-oriented programming (FOP) [9], improve SPL reuse and maintenance. Their composition mechanisms are intended to both explicitly capture the ways features interact and improve the reuse of existing modules. The expectation is that these techniques foster smooth integration of features and, therefore, are not detrimental to the SPL design stability.

Early research on framework integration focused on identifying implementation workarounds to achieve this goal (e.g [1]). Afterwards, some authors investigated how AOP (e.g. [10][11][17]) or FOP (e.g. [17]) could be used to extract features of legacy applications. Recent studies also analyze if a single SPL can be evolved with aspects or other means to smoothly accommodate changes (e.g. [4][7]). However, they do not evaluate how advanced composition mechanisms can support on-demand integrations of previously-designed SPLs. They do not investigate either the impact of adopting these mechanisms on the short- and long-term design stability of the integrated SPL.

This paper presents an exploratory, non-controlled study of on-demand integration of previously-designed SPLs. We compare the degree of reuse and stability achieved with AOP and FOP (Section 2.1) in this context. The AspectJ [8] and CaesarJ [3] languages were chosen to support the stepwise integration of features across three SPLs in the board game domain (Section 2.2). Each feature integration was identified and implemented on demand, meaning that none of the integrated SPLs were designed with the required changes in mind. CaesarJ was chosen as representative of FOP for two reasons: (i) its compiler proved to be robust during our pilot assessments, and (ii) there are public reports of their successful adoption in industrial projects. On the other side, AspectJ is the most popular AOP language. Our analytical aims were to address to following questions: (Q1) when the use of AOP and FOP promote higher reuse in typical SPL integration scenarios? and (Q2) is the higher degree of reuse achieved with a particular technique always converted to superior stability of the integrated SPLs? The results were derived from the application of previously-defined stability and reuse metrics [4] (Section 2.3). Our findings are summarized in Section 3, and final remarks are presented in Section 4.

2. STUDY SETTINGS
We started our study by selecting the AOP and FOP mechanisms (Section 2.1) to be used and assessed in recurring scenarios of SPL integration (Section 2.2). The employed metrics are presented in Section 2.3.
2.1 Target Composition Mechanisms

Composition mechanisms are used to combine modules, such as class and aspects, into larger pieces of software. To enable SPL integration on demand, composition mechanisms are used to combine independently-designed modules, which realize different features. In this study, specific AOP and FOP mechanisms of AspectJ and CaesarJ were employed. The mechanisms supported by AspectJ were: \textit{intertype declaration} and \textit{pointcut-advice}. The former makes it possible to add structural members – such as attributes, operations and classes – to other classes. The latter enables to combine aspect code at specific points (joinpoints) in the base program. An advice implements behaviour similarly to a method, but it is attached to a \textit{pointcut}, i.e. a group of selected joinpoints. The advice is implicitly triggered when each of the selected joinpoints are reached in the program execution.

As far as the CaesarJ language is concerned, two mechanisms supporting FOP were employed: \textit{virtual classes} and \textit{mixin composition}. All inner classes within a class (CaesarJ class) are considered as virtual classes, because they are handled similarly to virtual methods. This means that a class or a group of classes can be redefined in any subclass of the enclosing class. Then, virtual classes enable to treat inner, nested classes, polymorphically. This means that operation overriding and late binding can also be applied to inner classes. Using \textit{mixin composition}, different modules can be composed to build more complex modules without compromising the independence of each one. \textit{Mixin composition} is the way by which CaesarJ creates complete modules out of various parts.

2.2 Integrating the Target Product Lines

Three board game SPLs were used and integrated in this study: Shogi [8], JHess [13] and Checkers [14]. In total, the three SPLs have more 10KLOC, and implement 26 optional and alternative features that enable the derivation of several products. Checkers is an American checker whereas Shogi and JHess are chess games. All of them provide features to manage various functionalities for customizing the board (e.g. indicating movable pieces) and the matches between players (e.g. indicating player turns). New feature combinations make possible to generate several additional products, which could not be derived from the individual SPLs. The integration of such SPLs is also driven by the motivation of sharing non-trivial feature code, which would otherwise be replicated and maintained separately. For instance, there is a set of non-trivial features, such as save/load game, remote multi-players, undo/redo moves, which are candidates to be reused across all the SPLs.

\textbf{Feature Integration Scenarios.} Our study focused on four integration scenarios, with each of them leading to a new release of the SPL. These scenarios were chosen as they involve both fine- and coarse-grained features, i.e. those realizing either a few or many modules in the code. There are also integration cases involving functional features (e.g. customizations of pieces) and non-functional ones (e.g. persistence). As far as the non-functional features are concerned, they can be further categorized in homogenous or heterogeneous [5]. Homogenous features apply the same code at the different modules being affected whereas heterogeneous ones apply different pieces of code. All these diverse feature characteristics exposed the techniques to a wide range of feature integrations. It was not our goal to have an exhaustive list of integration categories [1]. Instead, we intended to observe to what extent the programming techniques support reuse and stability in recurring integration scenarios.

![Figure 1. Integrating Features from SPLs](image-url)

Figure 1 shows simplified feature models of Checkers (left) and Shogi (right); it illustrates the three first scenarios, which target the integration of features already implemented in these two SPLs. The first scenario consists in allowing the user to customize the colors of the Checker pieces. This integration is achieved through the reuse of the feature Customize Colors already realized in the Shogi game (Figure 1). The latter involves the rendering of pieces and handling of user interface commands. The second and third scenarios promote the integration of a Checkers feature into the Shogi SPL. The goal of the second scenario is to enable the feature “display movable pieces of the current player” being also part of Shogi game products. The third scenario involves a non-functional coarse-grained feature; it introduces the save and load operations in the Shogi game in order to store current piece positions at the board. Finally, the fourth scenario integrates the non-functional feature logging to Checkers from Shogi. The logging feature consists in storing the moves of the pieces, user customizations and methods execution.

\textbf{Realizing the SPL Integration.} To promote a fair evaluation of the composition mechanisms under analysis (Section 2.1), several steps were followed while implementing the SPL integrations, including: (i) refactoring of the object-oriented implementation of each SPL, (ii) applying a wide range of good programming practices (e.g. [16][18]) during such refactorings, and (iii) expose additional variabilities into original AOP and FOP SPLs following the guidelines defined in [10]. Theses steps were also important to guarantee that SPL design imperfections would not interfere in the results of our study. After these basic preparatory steps, a feature-oriented and an aspect-oriented implementation of the integrated SPL were generated. Each of them was incrementally modified to realize the four integration scenarios. A number of rules were enforced in the integrated SPLs, such as the constraint that non-mandatory features were encapsulated in specific family classes (CaesarJ) and aspects (AspectJ). There were also transformations rules of FOP to AOP and vice-versa described in [19]. They were used to avoid favoring a particular technique as the solutions were not developed at the same time. In fact, for each integration scenario, a programming technique was used to first achieve the integration; the implementation based on the other alternative technique followed the guidelines described in [19].

Code review and inspection were carried out in those implementations after each scenario for checking the adherence to good AOP and FOP design principles. The implementations were evaluated in terms of stability and reuse based on a set of previously-validated metrics (Section 2.3). A complete description of all the evaluation procedures is available at our accompanying website [15].

31
2.3 Reuse and Stability Metrics

In order to answer the questions Q1 and Q2 raised in the introduction, previously-validated metrics for reuse and stability were used. These metrics were chosen as they capture different dimensions of these attributes. They were also employed in previous studies of reuse and stability (e.g. [4]).

The degree of reuse was quantified by Reuse Percentage (RPercentage) and Reuse Rate (RRate) [4]. Reuse percentage evaluates the ratio between: (i) the code that remained unchanged for achieving the integration (e.g. the reused code that implements the integrated feature from the target SPL), and (ii) the total code of the SPL that incorporates the integrated feature. This measure is evaluated in terms of lines of code. Reuse Rate refers to the ratio between: (i) the number of modules from the target SPL that are reused (partially) during the integration (e.g. the modules which are part of the feature), and (ii) the total of modules of the original SPL after the integration.

The metrics Size of Change (CSize) and Scope of Change (CScope) were used with the purpose of quantifying the degree of modifications on code implementation. Scope of Change is the ratio between the number of modules affected by the change and the total number of modules of the original SPL, including the modules that were added in the integration. In fact, the changes may affect: (i) the modules from the target SPL which are part of the integrated feature, and (ii) the modules of the original SPL that are modified to accomplish the integration.

Size of change quantifies the lines of code (LOC) modified to fully implement the feature integration according to the equation below. The reasoning behind this metric is that the more code elements are changed to integrate a feature the more unstable is the existing SPL implementation. This metric enables us to analyze to which extent the integration also affected the other varying features and/or the core modules of the original SPL. 

The extent of the modifications is 6000, which is computed as follows: 50 x 30 + 100 x 15 + 150 x 20. Finally, the size of change is 20, which is obtained from the division 6000/300.

3. DISCUSSION

3.1 Reuse

The reuse was measured by computing the percentage of reused code and partial reuse. Figure 2 presents the reuse measurement for the four integration scenarios. Both AspectJ and CaesarJ presented almost the same measures for reuse percentage (RPercentage). There is no significant difference either between the number of reused LOC and the total number of LOC in the integrated SPL. In the integration scenarios, the code of the integrated feature was adapted in order to be incorporated into the target SPL. Therefore, partial code modifications were required when using both AOP and FOP techniques, compromising in turn the number of reused LOC. These alterations were mainly carried out to resolve syntactic conflicts, such as renaming operations. These modifications are further described in Section 3.2.

In relation to reuse ratio (RRate), CaesarJ achieved better results as illustrated in Figure 2. The CaesarJ SPLs decompose the feature code in more modules (than AspectJ) that collaborate to realize the overall feature behavior. In fact, there are evidences that the use of virtual classes contributes to a better modularization of the source code [4]. The minor superiority of CaesarJ was due to the use of virtual classes, which enhance the reuse of modules and their collaborations themselves. However, in general, the differences in favour of FOP, as supported by CaesarJ, were not expressive.

![Figure 2. Reuse Results](image)

3.2 Stability

In spite of the similar reuse results, there were significant differences in terms of stability. The superiority of FOP or AOP depended on the peculiarities of each feature integration scenario. The more modifications are required to integrate a new feature, the more unstable the system design is likely to become through the SPL evolution. Figure 3 illustrates the results observed in the integration scenarios in terms of change size and scope (CSize and CScope) for both techniques.

Bi-Directional Destabilization of SPLs. In the first scenario, the use of virtual classes leads to more stable implementations of CaesarJ. It required fewer changes than AspectJ, and their scope was narrower as well. CaesarJ promotes higher inheritance flexibility by supporting refinement in both family and inner class levels. Change size is significantly lower that AspectJ as the number of import statements is reduced. The reason is that family classes implicitly recognize all declarations present in their class hierarchy structure. On the other hand, aspects have to import all modules that will be affected, thereby increasing the number of modified LOC. This entails instabilities in both directions – i.e. in the feature code of both SPLs being integrated. The first scenario involves the integration of coarse-grained feature that embraces several modules of the target SPL. It involves various concerns such as user interface and rendering of pieces. The higher amount of changes in AspectJ implementations was due to the joinpoint model particularities; the feature composition is essentially tied to the syntax of the based program. This weakness was revealed by several modifications required to advise methods in the original SPLs. Conversely, CaesarJ concentrates all modifications in one
family module. This module encloses the classes that collaborate to refine the system behavior according to the integrated feature.

Figure 3. Stability Results

Privileged Aspects and Invasive Feature Integrations. The second scenario analysis suggests a superiority of CaesarJ in terms of stability. However, the gap with the results of AspectJ is lower when compared with the first scenario because the integration of the new fine-grained feature required the composition of fewer modules. Furthermore, the integration in CaesarJ also required a refactoring to expose elements with private visibility in the original SPL. These elements encompass modules for rendering a Shogi piece and moving attributes. In this sense, AspectJ provided more flexibility through the use of privileged aspects [8], which enable aspects to invasively affect the target modules. Then, this integration did not require refactoring to expose private fields in the AspectJ code.

Integrating Heterogeneous Crosscutting Features. The third integration involves persistence, a heterogeneous crosscutting feature. Different from the Checkers, the game Shogi includes several kinds of pieces such as horse, queen, and tower. The integration involves modules related to pieces, boards and user interface. Modules and methods are refined in the integration in order to both allow storing each piece type and position in the board and provide the user with interface elements to save and load the game. In AspectJ, it was hard to specify a set of joinpoints and to refine the collaboration among modules [5]. In this sense, it was necessary certain amount of effort to expose parts of both SPLs that would be intercepted. In fact, the high amount of modules that are modified during the integration were widely scoped and, as a consequence, refactoring operations have contributed to the less expressive AspectJ results.

In CaesarJ, the use of family classes and mixin composition supports late binding. However, in a virtual class, it is not known a priori the type of the referenced object as it can be rebound to a refined class in another family during the program execution. This peculiarity inhibits the use of virtual classes to save data information. The problem is that whenever an object is loaded from a stored file, its type is not known as it may be stored as a specialized type. In the third scenario, the CaesarJ module that stores the current state of the board was implemented in a Java class. This decision does not affect the current integration since it does not comprehend non-mandatory data. However, it led to an instability problem later in the fourth scenario.

Long-Term Instabilities of Feature-Oriented Design. The last scenario implies the integration of the logging feature into the Checkers SPL. This homogeneous crosscutting feature was previously implemented for the Shogi SPL. It is composed by three different types of logging: user customization, moves of pieces and method execution. The logging information might be stored. However, it must be possible to configure which logging types are persisted. In this sense, the CaesarJ implementation does not provide appropriate support for homogenous crosscuts. It leads to several modifications to implement the logging purpose of method execution. As various logging implementations are available, it is possible to generate several scenarios. Thus, it is necessary to storing all possible logging strategies, because the logging depends on the product configuration chosen. In addition, as the stored state of the board must be implemented in Java classes, it is necessary to manage a group of Java classes. As a result, CaesarJ has presented instabilities in the fourth scenario due to integration code implemented in the third scenario. CaesarJ got higher measures for change scope and size than AspectJ. The weaving process of AspectJ allows storing and saving objects without type modifications.

Figure 4. 4th Integration: AspectJ(top) and CaesarJ(down)

Figure 4 illustrates the implementations of both techniques for the fourth integration. In AspectJ, the use of "inter-type declaration", represented by dependency associations, allows variability in terms of the module that will be stored (DataBase). CaesarJ implementation relies on conventional inheritance to perform the same variability. Furthermore, family classes are needed to correctly instantiate the specialized type of the data module (PersistLogMoves, PersistLogUserCust and PersistBothLog). The virtual classes are alternatively composed to the family class responsible for persistence and bound via mixin composition.

4. CONCLUSION AND FUTURE WORK

Investigating alternative ways to improve productivity through SPL integration is becoming increasingly important. This paper reported a comparative assessment of AOP and FOP in the context of a stepwise on-demand integration of three SPLs. Our study confirmed and revealed potential benefits and drawbacks of AOP and FOP in recurring feature integration scenarios. Regarding question Q1 (Section 1), we observed minor reuse superiority for CaesarJ. Even though CaesarJ implementations promote more flexibility to reuse modules, the reuse is not necessarily reflected in terms of LOC. In fact, the measure differences were not significant.

As far as the question Q2 (Section 1) is concerned, our study confirms the expectation raised from previous studies [11][17] that FOP scales better when implementing heterogeneous crosscutting features. However, those studies only assessed the
modularity of features extracted of legacy applications without evaluating the effect on the reuse and stability of SPLs being integrated. The superiority of CaesarJ was thanks to the better support for on-demand refinement of feature collaborations. However, we found signs that the use of FOP mechanisms might negatively affect the stability of the integrated SPLs (Section 3.2). Therefore, our exploratory analysis should motivate the derivation of controlled studies to further investigate whether and how AOP and FOP mechanisms affect the long-term stability of SPLs.

As future work, we believe that this initial work can be further improved in many directions, including (but not limited to): (i) the extension of our study to embrace other emerging programming techniques, such as delta-oriented programming [6], (ii) the evaluation of trade-offs involving stability and other equally-important program attributes, such as undesirable feature dependencies, and (iii) the evaluations involving other SPLs in order to gather more evidence regarding our research questions.

5. REFERENCES