Feature Separation: An Approach for Product Line Evolution

Pan-Wei Ng

Abstract—**Although evolution is a critical aspect of software product line engineering, the body of knowledge surrounding it is still inadequate. The contribution of this paper is to show how the practice of feature separation practice addresses the evolution challenges, specifically in a telecommunications software product line case study. The main idea behind feature separation is to achieve a one-to-one relationship between features in the feature model to feature realizations in code, which keeps feature realizations separate in the code. This not only prevents the software product line architecture from deteriorating, but in fact improves it. To apply this in an industrial context, it is necessary to also take into account schedule pressures and legacy artifacts. Application into the telecommunications software product line case study shows significant productivity and architecture improvements.**

Index Terms—**Software product line, evolution, variability, feature separation, architecture, aspect orientation.**

I. Introduction

The basic idea behind software product lines is to take a set of core assets and to assemble product variants quickly [1]. The ability to deal with the variability between products, known as variability in space, is a critical aspect of software product line architectures with many approaches exist in academic literature [2], and more diverse methods exist in practice [3]. These methods often involve building a feature model that describes the variations between products.

Another important aspect is evolution, also known as variability-in-time, which is to software product lines. While evolution of traditional software engineering occurs in the maintenance phase, software product lines evolve throughout its lifecycle [4]. Elsner et al. [5] highlighted that the body of knowledge surrounding evolution is relatively inadequate compared to variability in space. Botterweck et al. [6] demonstrated how a set of change operators could take one version of the feature (requirements) model to yield another version of the feature model. However, research into the evolution of feature realization artifacts (e.g. source code) under a legacy setting is nevertheless limited. This paper is an effort to close this gap.

A. Objective of Paper

The goal of this paper is to demonstrate an approach, known as feature separation, to deal with software product line evolution in an industrial setting. This involves a telecommunications software product line (developed using C/C++) case study. We examine its resulting challenges pertaining to evolution, and how they are addressed using feature separation, which is based on our earlier work on aspect orientation [7], except that we now apply it to the context of software product lines and without using an aspect oriented programming techniques. Unlike [6], which is primarily about the evolution of feature models (i.e. the requirements space), our contribution is about the co-evolution of requirements and especially realization.

The key idea in our approach, which we call feature separation, is to attempt to align both the structure of requirements (i.e. the feature model) and the structure of the realizations (i.e. source code). By doing so, changes to successive releases can be separated from one another, and hence developed in parallel on a single mainline branch in the software version control system. This effectively transforms the problem from one that is variability-in-time to variability-in-space, where variability mechanisms are well-known. Moreover, it improves the cohesion of the realization, and thus has a positive impact on the software architecture.

Dealing with evolution is not easy, especially for our legacy large scale product line. It involves changes to both technical and management practices, which we will be discussing in this paper.

II. Context and Problems

Fig. 1 shows the layered architecture of the telecommunication software product line in our case study. It includes both hardware and software. Each block has its own core assets and variants. Different engineering departments with 100s of engineers are responsible for each separate block.

Each release introduces in the order of 100KLOCs of changes and takes a relatively long period (9 months) to develop and test on target hardware. Usually, the hardware is also upgraded as well. To prevent different releases from affecting one another, releases work, Fig. 2, on different branches in their version control system.

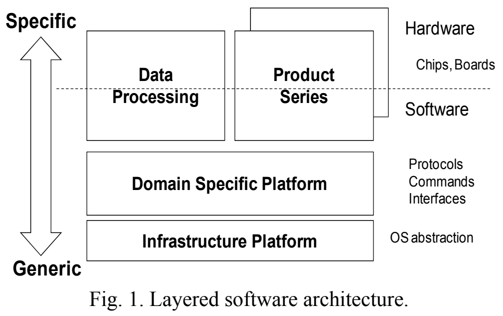


Fig. 1. Layered software architecture.

Manuscript received March 1, 2013; revised May 9, 2013.

Pan-Wei Ng is with Ivar Jacobson International, Singapore (e-mail: panwei@ivarjacobson.com).

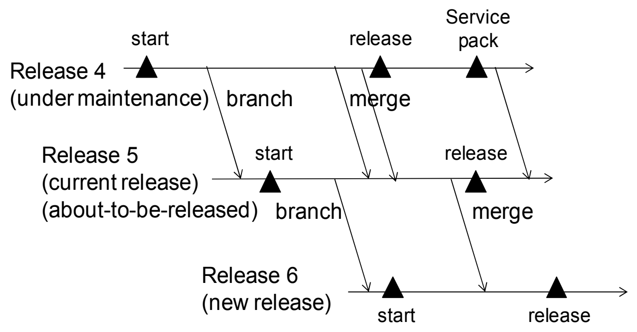


Fig. 2. Branching and merging

Ideally, the current release (e.g. Release 5) should always have high source code at all times. However, this cannot be guaranteed, especially since full verification requires hardware testing and integration testing of the various blocks in the layered architecture, which are at different stages of completion. Thus, the development of a new release (e.g. Release 6) does not start with a quality version, but instead has to merge changes and fixes from the previous version regularly. This poses significant effort and problems to the development teams because of the following:

1) The developer in Release 6 who performs the merge is usually not developer who made the code change in Release 5.
2) If a developer in Release 5 or Release 6 redesigns the same code, merging becomes complex. Very often, Release 6 has to re-implement the change. This makes developers very reluctant to take initiatives to improve their codes.
3) After merging, there is significant effort to conduct testing and bug fixing, both of which are error-prone.
4) Artifacts in a new release (e.g. Release 6) can be very different from that in the current maintenance (e.g. Release 4). Thus separate teams are needed, which is an overhead.

Merging, a fundamentally non-value-added work is error-prone and consumes significant resources and time. The observation is that if no release branching occurs, then release merging can be eradicated. Thus, it was suggested that all releases ought to work on a single mainline. However, there are two important barriers:

1) Examining the codes shows the presence of tangling and scattering. The realization of requirements (i.e. features) are scattered across different parts of the software, which is built by different members. There is very little assurance that changes originating from one release do not affect the behavior of the previous release if releases were to work on the same mainline.
2) Release managers like the use of separate branches because it allows changes in each release to be completely isolated and they feel they have good control of the release schedule and quality. Thus, they oppose the idea of working on a single mainline.

One possible solution is to ensure that there is no overlap between successive releases. However, due to the long time frame to stabilize the hardware, which is developed in parallel, and market needs for large number of features, the overlap in this particular software product line cannot be eliminated.

III. The Solution: Feature Separation

Our recommended solution taken was to preserve the separation of features (and their variants) in requirements all the way to code and test. This allows the development different releases on the same mainline without affecting each other. Our approach is known as feature separation. Its basic idea is to preserve the separation of features seamlessly to code and test, which was first discussed in [7]. The contribution of our paper is to demonstrate how this solution is applicable on a large scale legacy system – specifically a legacy software product line evolved under tight development schedules.

A. The Basic Idea of Feature Separation

Fig. 3 shows a traditional development approach, whereby a product version requires changes to 2 existing features. These two features are realized by different parts of the source code (e.g. different folders or files). Regardless of the size of the change, the system has to be tested.

The many-to-many relationships (i.e. scattering) depicted in Fig. 3 create a number of problems. Firstly changes are amplified necessitating extensive testing effort. Secondly, the complex relationships pose a traceability overhead, which the teams are reluctant to pay because they have to test the whole system anyway. Thirdly, inexperienced developers who have yet to learn the relationships tend to introduce bugs into the product.

Fig. 4 shows an alternative approach using feature separation. A product version is still free to select different feature variants in the feature model. However, features, source code and tests are structured such that they are aligned to one another, thus reducing scattering. The ideal case occurs when there is one-to-one relationship from features to code and test.

The application of feature separation requires:

1) A good feature model structure whereby source code and test structures can be aligned. This is achieved through the use of several feature patterns.
2) A systematic approach to align source code with the feature model structure. This is achieved through feature separation mechanisms

In addition, it is important to have strategies to deal with legacy code and ways to manage the evolution of the software product line (i.e. from feature model, to source code and test.)

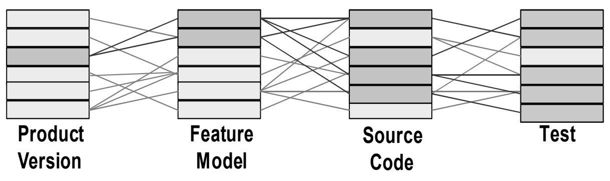


Fig. 3. Traditional development.

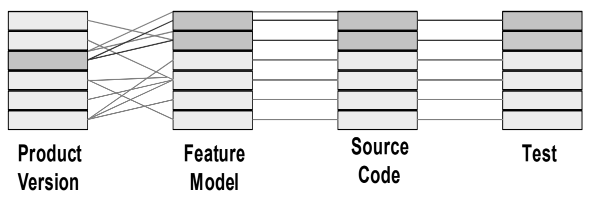


Fig. 4. Feature separation

B. Feature Separation Patterns

As mentioned, feature separation requires establishing well structured feature model. This feature model can be constructed using several feature patterns, which are categorized as follows [7]:

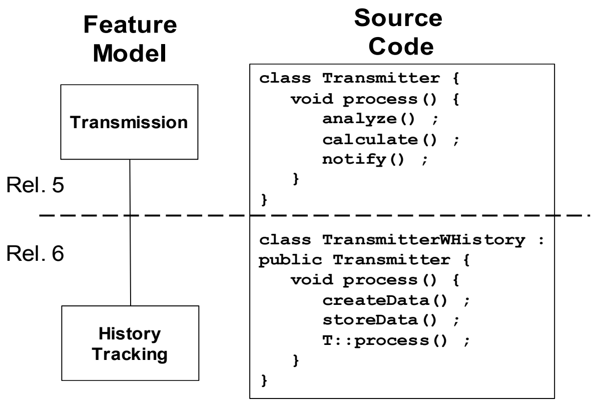
1) Peer features – These are features that have separate and distinct from the user point of view, but their realization touches the same pieces of code. In general, there would be mechanisms like dispatchers to invoke these distinct features. There would be reusable elements that are invoked by the features.
2) Extension features – These are features that are enhancements of existing features. In the simplest case, it requires that existing code invoke the extension code. In more complicated case, extension features may decorate (modify) the behavior of existing code.
3) Feature interactions – This is a combination of peer and extension features whereby two features by themselves are separate and distinct. However, when both features are activated, they modify each other's behavior. One possible solution is to treat them as extensions of one another. Alternatively, a coordination layer is added between them.
4) Framework features. All software systems run on top of some underlying framework. Framework features are generally crosscutting. They are invoked just before invoking each functional feature. Thus, this is a special case of extension features by merely extending the framework itself.

The idea is to keep the different kinds of features separate at requirements time. In practice, this needs considering the design limitations as well.

C. Feature Separation Mechanisms

Once an initial feature model exists, evolving a software product line is merely a simple act of attaching new features (according to the patterns above) to the feature model. While attaching a new feature node, the development team will also attach a new corresponding realization element.

Fig. 5 shows a simple example whereby a Transmission feature in Release 5 is being enhanced by a history tracking feature in Release 6. Through simple inheritance, history tracking in source code is kept separate from that of the transmission feature. Separate factory classes in a product configuration folder instantiate the required class for each release. With such a seamless separation, the resulting from requirements in Release 6 are isolated from that in Release 5.



```
class Transmitter {
   void process() {
      analyze() ;
      calculate() ;
      notify() ;
   }
}
```

```
class TransmitterWHistory :
public Transmitter {
   void process() {
      createData() ;
      storeData() ;
      T::process() ;
   }
}
```

Fig. 5. Example of achieving feature separation through object inheritance

Inheritance is but one of the many techniques to achieve feature separation. Issues involving multiple-inheritance could be solved through C/C++ templates, and mix-ins. There are other design patterns and idioms that available.

D. Dealing with the Legacy Code

As mentioned earlier, our software product line case study involves significant legacy code that limits the ability to apply feature separation directly. Some artifacts (i.e. source code) through years of change have degenerated significantly, exhibiting severe tangling, scattering and duplication. Developers in this product line case study needed guidance to balance between achieving ideal feature separation and meeting their schedule deadlines.

As such, we identified a number of evolution strategies categorized according to whether the existing artifact is well-structured (Fig. 6) or poorly structured (Fig. 7). In these figures, the shaded part represents the artifact for the to-be-released version (e.g. Release 5), and the white part shows the change being introduced by the new version (e.g. Release 6). A rectangle shape denotes good structure and an odd-shape denotes poor structure.

Our identified evolution strategies are described as:

1) Unstructured insertion – This is essentially squeezing new code into the existing code. This is forbidden since it turns good code into bad code. It is a habit which developers need to break. However, at times when schedule pressures are tight, this is inevitable.
2) Structured insertion – This is structured way of adding new code, but it results in code bloat and god classes. Though not recommended, it is still not as bad as the previous case.
3) Unstructured extraction – This creates a new function/class to introduce the change in the new release.
4) Structured extraction – This is a more elaborate form than and is used when interfaces are well-understood or when multiple variants are conceived.
5) Partial Re-Design – This re-designs part of the code and keeps the enhancement separate.
6) Complete Re-Design – This is a complete re-design, and is most risky. This requires both the developers of both releases to work together.

Developers work with release managers to determine which of the above strategies to use.

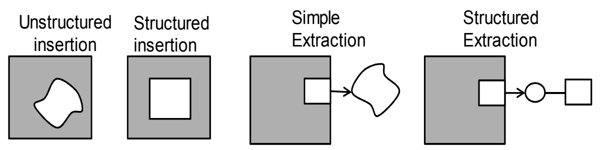


Fig. 6. Evolution strategies (when existing artifact is well-structured)

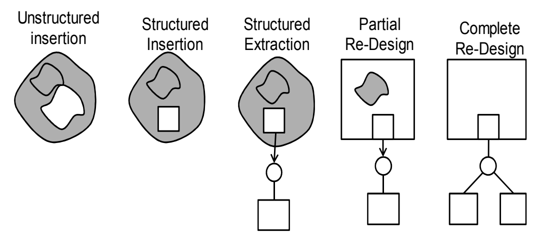


Fig. 7. Evolution strategies (when existing artifact is poorly-structured)

E. Managing Feature Development

As mentioned earlier, the manager of the current release (Release 5) who is about to be released does not like to have his work subject to changes which he could not control. One the other hand, the manager of the newer release (Release 6) favors mainline development because he can escape the merging work.

Release 6 impact on Release 5

Feature	Current State	Modification Approach	Risk
F1			
F2			
F3			
F4			

Fig. 8. Managing risks in evolution

To help alleviate current release (Release 5) manager's concerns, Release 6 manager shares his list of features, which has information about the current state of the feature – whether the codes to implement it is structured or unstructured, what is the modification strategy, as explained earlier, and the risk (high, medium low).

With this information, both release managers are able to schedule when features are being developed. The idea is prevent risky features from being introduced into the mainline near any release date. Risky changes are schedule way before a release date or after a release date. Usually, large risky changes are broken down first.

F. Measuring Feature Development

During the evolution of the product line, measures are tracked to ensure smooth development and to detect problems that may occur. These measures are depicted in Fig. 9.

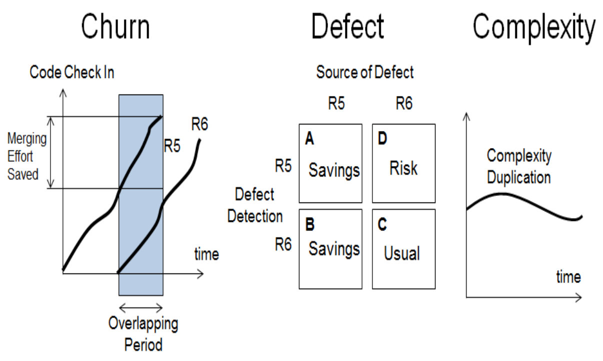


Fig. 9. Managing the progress

Churn measures the amount of code changed by different releases. It is used to check the actual period of overlap and amount of code that has changed during the overlapping period. The code churn by the current release represents savings since this amount of code no longer needs merging.

Defect rates measure the quality and the effectiveness of feature separation by identifying who detects defects versus who is the source of defects.

1) Quadrant A – This is savings for Release 6 since Release 6 no longer needs to merge the fixes.
2) Quadrant B – This is savings for Release 5 since merging from Release 6 to Release 5 is also eliminated.
3) Quadrant C – This is business as usual if Release 6 detects its own defects.
4) Quadrant D – This is a red light when Release 5 has defects introduced by Release 6.

Complexity is about measures of tangling, scattering and duplication. Through feature separation, files would become smaller and the cyclo-metric complexity of each method would go lower simply because tangling is reduced. Code duplication would be reduced as well, but sometimes developers may create to copies of the same code under schedule pressure.

IV. The Experience and Results

We introduced the concept of feature separation to the teams in the Data Transmission block (see Fig. 1), who had about 80 developers assigned to Release 5 and the same number of Release 6. The teams were distributed across three different cities to be in close proximity with their product variants.

The overlapping period between Release 5 and 6 occurs was about 4 months. During this time, Release 5 introduced 200 KSLOC in new features, and fixed 1000 defects. This amounted to savings of about 50 man months since merging was eliminated. However, Release 5 detected 4 defects introduced by Release 6. This was analyzed in detail and it was found that the reason was because the corresponding features were not well separated in code.

Overall, the development teams were very encouraged by the results. Developers now put more emphasis on design considerations, as opposed to merely implementing features. Feature separation spread throughout the organization quickly and was adopted by the entire software stack (see Fig. 1) across various departments.

Feature Separation is not the only practice which the teams use. Both releases 5 and 6 had adopted an iterative style of development. They also have a well fortified continuous integration environment in place to detect low level problems quickly, which gives developers feedback that their features are separated adequately in code. This reduces risks and is in fact is an important pre-condition for introducing feature separation.

V. Conclusion and Future Work

We have just shown how feature separation is used to deal with the evolution of a telecommunication product line. The results were promising and well received by the organization.

In our case study, human resources were organized by releases. Now that the teams are working on the mainline, there is no longer a necessity to split human resources by releases. A new way to organize developers would be needed. Moreover, testing for different releases would also be conducted in parallel and on the mainline. This necessitates changes to test planning. Thus, there is still a lot of room for research and improvement to cover different aspects of product line evolution.

REFERENCES

[1] K. Pohl, G. Bockle, and F. V. D. Linden, *Software Product Line Engineering*, vol. 10. Springer, 2005.

[2] L. P. Chen, M. A. Babar, and N. Ali, "Variability management in software product lines: A systematic review," in *Proceedings of the 13th International Software Product Line Conference*, Carnegie Mellon University, 2009, pp. 81-90.

[3] J. Bosch, "Maturity and evolution in software product lines: Approaches, artefacts and organization," *Software Product Lines*, 2002, pp. 247-262.

[4] Pussinen, Mika, "A survey on software product-line evolution," *Tampere University of Technology*, 2002.

[5] C. Elsner, G. Botterweck, D. Lohmann, and W. S. Preikschat, "Variability in time—product line variability and evolution revisited," 2010.

[6] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski, "Evofm: Feature-driven planning of product-line evolution," in *Proceedings of the 2010 Icse Workshop on Product Line Approaches in Software Engineering*, ACM, 2010, pp. 24-31.

[7] I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*, Addison-Wesley Professional, 2004.

Pan-Wei Ng received his Ph.D. from Nanyang Technological University. He is a software engineering coach and advisor with large organizations in Asia Pacific helping them with scaled agile transformations, product line requirements, architecture and testing. He is the author of "Aspect Oriented Software Development with Use Cases" and "The Essence of Software Engineering: Applying the SEMAT Kernel".