Eliciting Abstractions from a Software Product Line

Charles W. Krueger
BigLever Software, Inc., 10500 Laurel Hill Cove
Austin TX 78730 USA
ckrueger@biglever.com

Dale Churchett
Salion, Inc., 720 Brazos St., Ste. 700
Austin TX 78701 USA
dale.churchett@salion.com

Abstract. The effectiveness of a software product line depends on the effectiveness of the abstractions used to manage it. Salion uses a combination of technology and vigilance to elicit emerging abstractions in its software product line. The result is greater commonality, more concise variations, and faster time-to-market for new product family members.

1. Introduction

The effectiveness of a software reuse technique depends on the effectiveness of the abstractions employed.[1] Since software product line approaches gain their benefit from large-scale software reuse, the effectiveness of a software product line will likewise depend on creating and maintaining effective software product line abstractions.

Salion uses technology and vigilance to aggressively search out and refactor abstractions in its software product line. As a result, Salion optimizes its software product line through greater commonality and more concise variations. This provides faster time-to-market for new products, lower development costs, and higher software quality.

One example of eliciting software product line abstractions is described in this paper. This case identifies an emerging abstraction in a variation point that reduces the lines of variant code per family member from an average of 1600 to 250.

2. Technical Approach for Eliciting Abstractions

During the normal evolution of a software product line, variations are often introduced to provide greater flexibility in the product line or to expand the scope of the product line into new areas. This generally leads to both the introduction of new variation points and the growth of variation inside of existing variation points.

This tendency towards increased variation reduces the ratio of common software to variant software and thus reduces the effectiveness of software reuse in the product line. That is, there is a natural entropy, or tendency for divergence, that occurs during product line evolution.

To counteract this entropy, vigilance is required to constantly search for existing or emerging abstractions in the variation points. Once identified, these abstractions enable the variations to be refactored into greater levels of common software and more concise variant software, thereby increasing the levels of reuse in the product line.

Experience at Salion has shown that appropriate software product line technology can help to elicit abstractions. The first technology example is software product line infrastructure that clearly encapsulates *variation points* in the source base of the software product line. For example, Salion uses the GEARS software mass customization infrastructure from BigLever Software.[2,3,4] GEARS provides explicit constructs that encapsulate and localize source file variations in a software product line. The lead architect at Salion routinely scans through the variation points in search of existing or emerging abstractions. Without the explicit and localized encapsulation of the variation points, it would be considerably more difficult to know where to look and what to compare in the search for abstractions.

The second type of technology that aids in the elicitation of abstractions is software comparison. A simple example is the conventional UNIX *diff*. It can be used to search for cut-copy-paste *clones* within a variation point. More advanced structural abstractions can be found using a tool such as the *CloneDoctor* from Semantic Designs.[5] Salion uses this tool to search for structurally similar abstractions both within and among different variation points and common software. CloneDoctor is able to find abstractions from similar but not identical software fragments that often result from cut-copy-paste-modify during development.

Experience at Salion indicates that at least one engineer must be committed to the task of eliciting abstractions from variation points. Many engineers developing software in variation points will not have the time, skill, experience, or interest to search out abstractions. Without vigilance by skilled architects or generalists, entropy will take hold within the variation points and the effectiveness of the software product line will degrade.

3. Case Study

Development on Salion's product suite began with little customer input. The system was designed based on knowledge gathered by initial market research, customer demos and industry experts. Consequently the software design had to be robust in the face of certain change. Salion's development team adopted a component-based development practice, an agile development process, and a reactive software product line approach from the start.[6,7]

As the first few customers were deployed from the product line, the variation points of the system began to stabilize. Analysis showed that previously unseen abstractions were emerging within the variation points.

In one example, a developer had created a variation point that encapsulated variants of a façade object used by user interface developers. The façade handled complex logic revolving around versioning, database operations, and view helper objects. The first façade variant implementation required 1600 lines of code. After realizing that much of the code would be identical for each variant implementation, he began to search for an abstraction. By applying diff to two variants, the common code was moved into an abstract superclass outside of the variation point and the façade variants were implemented as subclasses in the variation point (see Figure 1). The Template Method design pattern was applied to the superclass so that subclasses were required to implement only one method, but if required they could override three more.[8]

Because unit tests were already written for the two façades, refactoring the abstraction out of the variation point took only two days (including testing). The developer and a co-worker (who had been assigned to the next façade variant implementation) conducted a pair-programming session. The benefit of pair programming on the façade variation point refactoring task proved to be invaluable. First, the co-worker learned the design strategy of the superclass and the responsibilities of the subclass. Second, bugs were spotted earlier and more unit tests were written as needed.

Once the abstract façade was implemented, the first façade variant was reduced from 1600 to 600 lines of code (LOC). The second façade was implemented in 30 LOC (constructors were all that were needed); the third façade was implemented in 80 LOC and the fourth in 400 LOC. By abstracting code from a set of variants in a variation point, a usable and practical framework emerged.

Inevitably, as soon as the product line was put into production, a bug was found. The source of the bug was traced to the abstract class, so a single fix was automatically picked up by all product flavors. If the same bug had been found in the original implementation, the fix would have been required in each variant (assuming the developer remembered to do them all).

In general practice, the Salion software architect is able to continually monitor the software product line for emerging abstractions in the variation points, first by visual inspection and second by applying code diffs and clone detection technology. Because GEARS isolates variations in subdirectories, variations for a common abstraction can be easily and efficiently elicited.

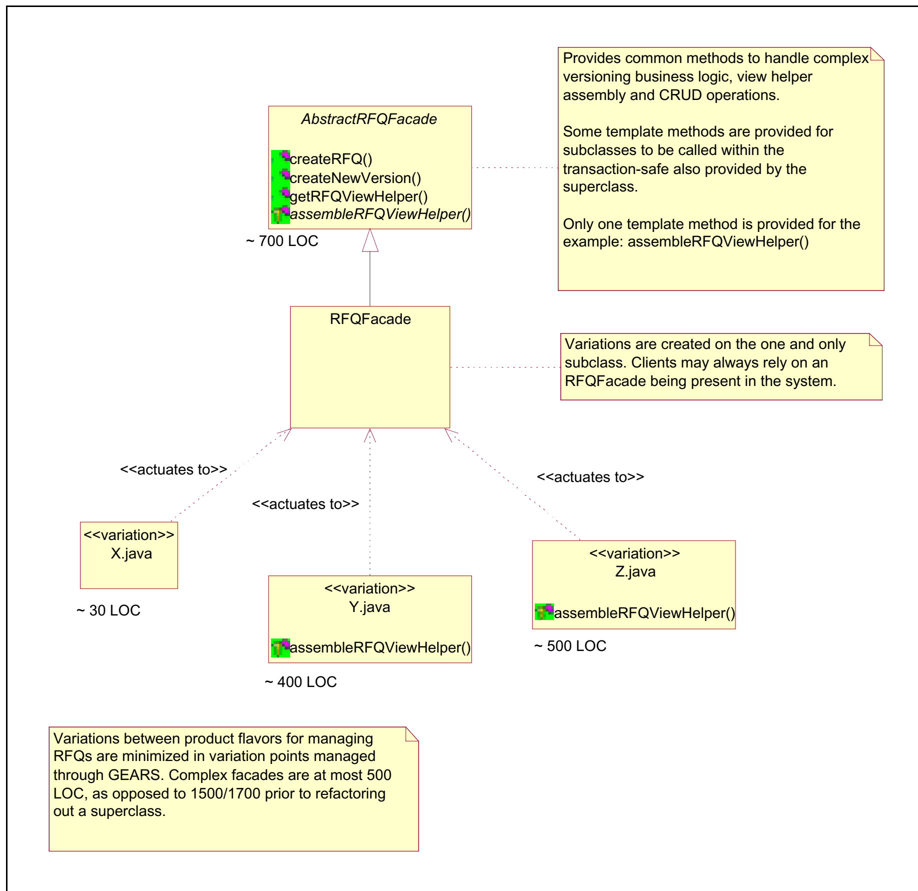

Figure 1

4. Conclusions

Experience at Salion has shown that the engineering team must tightly control the natural entropy present during software evolution in order to maintain the effectiveness of their software product line. One way to control this entropy is to constantly search out and elicit emerging abstractions in the variation points of the product line. The following technology and techniques in combination can accomplish this:

- Encapsulated variation points, such as provided by GEARS, to clearly identify where to search for emerging abstractions
- Diff and clone detection for mechanically identifying potential abstractions
- Constant vigilance and a development process that promotes constant improvement through refactoring
- Traditional object-oriented design skills and patterns

References

1. Krueger, C. *Software Reuse*. 1992. ACM Computing Surveys. 24, 2 (June), 131-183.
2. Salion, Inc. Austin, TX. www.salion.com
3. BigLever Software, Inc. Austin, TX. www.biglever.com
4. Clements, P. and Northrop, L., *Salion, Inc.: A Case Study in Successful Product Line Practice*, Software Engineering Institute, Carnegie Mellon University, Technical Report in progress.
5. Semantic Designs, Austin, TX. www.semanticdesigns.com
6. Buhrdorf, R. and Churchett, D., *The Salion Development Approach: Post Iteration Inspections for Refactoring (PIIR)*, Rational Edge, www.therationaledge.com/content/mar_02/m_salionDevelopment_rb.jsp, March 2002.
7. Krueger, C. *Easing the Transition to Software Mass Customization*. Proceedings of the 4th International Workshop on Product Family Engineering. October 2001. Bilbao, Spain. Springer-Verlag, New York, NY.
8. Gamma, et.al., *Design Patterns*, Addison-Wesley, 1995.