Structured Variation Management in Software Product Lines

Felix Bachmann
Software Engineering Institute
fb@sei.cmu.edu

Linda Northrop
Software Engineering Institute
lmn@sei.cmu.edu

Abstract

Variation management is one of the key challenges associated with software product lines. Much is written about this topic. There are methods and tools available, home-grown and commercial, which attempt to make handling product line variations practical.

The focus of these methods is on supporting software development; that is, handling variations in the core assets. However, there are non-code core assets and they also have variations. For example, there are marketing roadmaps, requirement specifications, test cases, process definitions, and all kinds of documentation including user guides, to name just a few. These core assets also have to implement the required variations of the product line and need to do so in a consistent way or there will be incompatible implementations and resultant inefficiencies.

The approach to variation we are suggesting takes an organization-wide perspective, covering the entire gamut of core assets not just the code. We describe the definition, implementation, and management, including tools support, of a few essential variations throughout the organization to make handling product line variations more efficient and effective.

1. Introduction

According to the Carnegie Mellon® Software Engineering Institute (SEI), a software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [7].

Part of the prescription is the way in which variations across the products in the product line are handled. Variations have to be accommodated by the whole set of core assets (code and non-code) and have to be planned very carefully to support efficient creation of new products in the product line. When a variation is needed, such as adding a new feature, the goal is to isolate necessary changes, to as few places as possible. Without a doubt, some of the primary core assets are the software architecture as well as the design and implementation of the components that comprise the software used in product line. This, in many organizations, leads to the belief that dealing with variation is purely a technical problem and therefore the responsibility of software developers.

However, the introduction of variations into a product line is really a business proposition. It is typically driven by the market needs. For example, an organization developing entertainment systems for cars may decide that they need a low-end system containing only a radio as well as a high-end system additionally including a CD changer and voice control. In this case, the CD changer and voice control would define variations to be accommodated by the product line, while the radio would be a common feature. To be profitable, the development organization has to ensure that building products with or without CD changers and voice control can be done easily and efficiently. We call these kinds of variations “essential variations” [8]. Every part of the organization that produces core assets must support the essential variations. For example, the group producing the user’s guide and the group concerned with testing and integration are just as concerned with the essential variations as those writing the code.

To permit efficient product production, the essential variations must be interpreted throughout the organization in a consistent way. For example, if the entertainment system includes a CD Changer, then the user’s guide must describe this changer. It is a challenge for the organization to ensure that everyone has the same interpretation of the meaning of the variations so that the core assets are created in such a way that the final product is within the market definition.

In addition to the essential product line variations we also define “local variations.” These are variations that are introduced by a group within the product line organization responsible for a specific type of core asset. These variations are geared towards supporting
the product building process for this particular type of core asset. For example, the technical writers may decide to have a different font and/or different images in the user’s guide if the guide comes with a luxury car or not. Or the software developers may decide to define the ability to choose between code libraries of different vendors as a local variation.

It is useful to have some means to analyze the definition and use of essential and local variations throughout the product line organization. Analysis tools using a centralized database would be appropriate, but the larger the product line organization, the more it behaves as a “decentralized system with a common goal” [10]. Adding a central organizational unit with the task to collect and maintain a repository of information contributed from all involved departments and to ensure a consistent implementation throughout the organization carries a high risk of failure. Organizational groups work for their own interest and do not necessarily accept guidelines from a central organization if those guidelines do not have obvious benefits for the groups.

For product line variation management to be an accepted practice it needs to be distributed. Every group within the product line organization uses and will contribute to variation management if the right incentives are in place. What incentives those might be is the subject of another area of research and not addressed by this paper.

The goal of the work described in this paper is to take the distributed nature of variation management into account and provide a first version of a method with tool support that would minimize the additional work needed to explicitly specify variations. The method should also provide benefits for local groups as well as the whole product line organization by avoiding rework because of misaligned variations.

2. Essential Variations

One way of making variations in a product line manageable is to structure them in such a way that every role in the organization, such as developer, technical writer, tester, only has to be concerned about variations that affect their role. The fewer variations required the easier it is to manage them. But there is a tension between this goal and the organizations needs. Every group within the organization has the tendency to increase the number of variations to support their particular needs.

Software developers especially seem to tend to define many variations (necessary or not) to prepare for an unknown future. Very quickly the number of variations increases and the benefit of having only a few isolated changes disappears. The more variations that are introduced, the more likely it is that the development organization loses control over the software because no one understands the effect of all those variations. In large embedded systems it is common to have more than 2000 defined variations. It is a daunting task to testing such a system. It is only possible to test a specific instance (a specific set of variation parameters) thoroughly and therefore the testing activity cannot benefit from the commonality in the product line.

Variations need to be grouped to be manageable. They need to be grouped according to the type of core assets produced by a role within the organization. For example, the marketing department is most likely not interested in the variations the testers of the QA department introduce, but they are interested in variations that increase the chances for product placement in the market, such as flexibly exchanging the display of a device.

Some groupings of variations need to be coordinated to achieve a specific product variant. Those groups of variations are then organized into a hierarchy. The top group in this hierarchy contains all the variations that require coordination between the different parts of the organization. The next lower level contains all the variations that are of interest for a specific part of the organization and only require internal coordination. If required, a specific part of the organization may decide to further break down the hierarchy. For example, the development department may decide to have operating system specific groups of variations, if a product is implemented as a WINDOWS as well as a LINUX version. This creates a structure similar to the one shown in Figure 1.

The essential variations are at the top level of this variation hierarchy. The hierarchy also defines the responsibilities of the groups assigned to implement the variations. Each group is responsible for their own “local variations” as well as for any higher level variations. For example, the technical writers are responsible for whatever local variations they introduce to make their work easier as well as for appropriately addressing all essential variations.

There is also the responsibility for defining the variations. Since a smaller number of variations is easier to manage, it is a good idea to have only very few people who are permitted to define variations, with the goal to have just enough variation but no more. This is especially true for the essential variations. Those variations have to be managed consistently across the whole organization, which can be a daunting task. The “What to Build” pattern
described in [7] gives some advice on who should be involved in the defining the essential variations. Common artifacts used for describing essential product line variations include product roadmaps, feature models, and feature lists.

3. Manage Essential Variations

For the remainder of this paper we focus on the management of essential variations. Although, all the techniques and tools we mention for the management of essential variations can also be applied recursively to the lower levels of the variation hierarchy.

There are two important goals for managing essential variations: Keep the number of variations low and manage them consistently throughout the organization.

We already mentioned that one approach to keep the number of variations low is to assign someone the responsibility of defining them. We now focus on the second goal, how to manage the variations consistently throughout the organization.

This sounds easier than it is. A consistent implementation of essential variations requires first determining an overall approach to handling the variations and then communicating both the variations and the overall approach to all parts of the organization, providing feedback about individual variation implementation so that consistency checks can be performed. The communication activity should be automated as much as possible to reduce effort but also to reduce errors when collecting information about implemented variations. An infrastructure similar to the one depicted by Figure 2 is a candidate solution for an organization-wide variation management system. A proof-of-concept implementation of such a system based on the variation model definition in [8] was implemented as a thesis for a masters degree.[9].

3.1. Communicate the Essential Variations

Essential variations have to be understood in the same way throughout the organization. Every group that produces some core asset depends on that knowledge. It is very expensive to find out at the end of the development of a product that some of the artifacts that are part of product deployment, such as the user’s guide, are not aligned with the product.

At the very minimum, every essential variation has to have a unique name and a definition. For example: A variation could have the name “CD Changer” and the definition: “… 5 disk changer from famous company, model CD34X …” or “… a CD changer can be one of the following models …”. Everyone in the organization has to know what “CD Changer” means so if someone would ask the question: What did you do to support products with or without a “CD Changer”, the answer is clear.

Other information can be present, such as which products those variations apply to and what the dependencies are to other variations. An example of other information would be: Voice control can only be included if also microphones with noise reduction are available, assuming “voice control” as well as “microphone” are also essential variations. A more detailed description of possible information about variations can be found in [8].

Commonly used ways to communicate essential variations are product roadmaps and feature models [1], [2], [5]. All of these in some way show the features of different products and provide some indication of what varies and what is common. There are a number of available feature modeling tools [4], [3].

Features themselves are not necessarily variations. Features that are in every product are commonalities, not variations.

We suggest using a definition of essential variations based on feature models because they usually provide a method and notation sufficient to successfully setup variation management. The proof-
of-concept implementation uses Antkiewicz’s Feature Modeling Plugin (FMP) [4].

Figure 2. Variation Management System

3.2. Providing Feedback

No matter how well the essential variations are communicated throughout the organization, there will usually be inconsistencies for a variety of reasons. There could still be misinterpretations, someone just forgot to check for variations, or some of the additional information for variations, such as who is actually deciding on the variant to use (binding time), have been ignored. For example, the plan could have been that the system administrator can decide on making a feature available or not by changing a configuration file. This most likely would require someone to create a process to follow and/or another developer group to create a tool that helps to manage the configuration file.

To find those inconsistencies early on, some feedback mechanism is required. In the simplest case, it could be a list mapping the variations introduced into the core assets to the essential variations. Such a list could be a text document, a spreadsheet, a database, etc. The list of all groups involved in the product development could be fed into the variation management system to perform consistency checks.

Manual processes are generally expensive and error prone. It is much more efficient if the feedback process is automated. That raises the following question: “What needs to be done to automatically extract variation information from all the different kind of core assets and make that information available to the variation management system?”

4. Implementing a Variation Management System

To achieve automation, three actions have to be performed:

- The parts within core assets that vary across different products need to be marked explicitly
- That information has to be extracted and converted into a common format
- All the extracted information has to be merged into the variation management system

Different core assets may be created using different tools. If the core asset is a document, then it was created using some text editor; if it is a code file, then it was created by some kind of language editor; etc. Some of those tools already have the ability to specify variable parts, such as conditional paragraphs, in a document. Other tools may not have this ability. One approach that always can be applied is to add some commented pseudo code into the core assets.

The exact definition of what needs to be added to describe some variable part in a core asset depends on the needs of the organization. Beside a name and a description it can include binding time, conditions on which specific variants depend, the skill set required to make the variant, the list of available variants, and so on. See [8] for a more complete list of variation attributes. Figure 3 shows an example annotation for Java files used in a proof-of-concept implementation. Another approach that can be used for code core assets is the XML-based Variant Configuration Language (XVCL) [6].

```java
/**
type: variable part
name: security component
description: adds an additional network security filtering layer
condition: highSecurity
binding time: compilation
execution skill: original developer
variants: empty, lowCost, highLoad, highSecurity
*/
NetworkServerComponent.setSecurityComponent( SecurityFactory.createEmptySecurityComponent()); // see SecurityFactory for more alternatives
```

Figure 3. Commented pseudo code in java file

Core asset developers have the responsibility to ensure that all variation information is included in the core assets. For core asset developers this should not be anything new. The description is an implementation of the attached processes as described in [7] and therefore should already be done.
in some fashion. What is added is that all core assets developers across the organization should do this in such a way that the information can be extracted automatically.

This gets us to the second activity: automatically extracting the variation information from the core assets and exporting it in a common format. Two approaches are possible. The organization might use a tool for creating core assets that allows adding extensions, such as IBM Rational Software Modeler. The extension mechanism can be used to extract the variation information.

If a tool is used that cannot be extended then a parser that understands the core asset format can be written to extract the information. If neither approach works then a manual approach to produce the variation information can also be used. Although a manual approach is error prone and far from being ideal, it is a practical compromise for an organization to get started with variation management and hopefully help the organization discover that an automated process would be a valuable investment.

In any case, the result of the extracting activity is the variation information included in the core assets in a common data format. XML files are a possibility and have been used in the proof-of-concept implementation. An example parser output is shown in Figure 4.

```
<coreAsset name="KWIC_SPC">
  <internalInput name="INPUT_METHOD"/>
  <internalInput name="MEMORY_SAFETY-LEVEL1"/>
  <variablePart name="INPUT_METHOD" input="true" generic="false">
    <variant name="FILE"/>
  </variablePart>
  <variablePart name="x_CircularShift" input="false" generic="false">
    <variant name="default">
      <variablePart name="CIRCULARSHIFT_INM_METHODS" input="false" generic="false">
        <variant name="replacesDefault"> ...
        </variant>
        <process description="Automatic: a variable part from a referenced file with the same ID is adapted."/>
      </variablePart>
      <process description="Automatic: content of xvcl file with the same name is processed and included here."/>
    </variant>
    <variablePart name="x_Input" input="false" generic="false">
      <variant name="default">
        <process description="Automatic: content of xvcl file with the same name is processed and included here."/>
      </variant>
    </variablePart>
    <condition description="This asset is used as a variant within the referenced variable part. Can not exist on its own." onName="KWIC_SPC"/>
  </variablePart>
</coreAsset>
```

Figure 4. XML output of a parser

This leads us to the final step, the merging of all the output into the variation management system. Merging in this context means connecting all the pieces of information together to be able to answer questions such as: “Are all the essential variations implemented in the core assets?” or “What core assets are involved in realizing a specific variation?”

The merging process creates an overall variation model connecting all the information bits together. The variation model conceptually looks like the depiction in Figure 5. Although the proof-of-concept tool includes a graphical representation of the resulting variation model using the GEF and EMF Eclipse plug-ins, that representation is not usable in practice because the amount of information presented makes the graph unreadable.

We suggest inserting the variation information into a database and using standard query tools to do the analysis. One interesting analysis we did after the creation of the variation model was to show all the variations in all the core assets that depend on a required feature. A required feature means that this feature is in every product and therefore is not a variation. If there are any variations in core assets that would depend on such a feature then those variants always have to be included and therefore there is no reason for actually making them a variation. This is either a result of an over-engineered solution or a result of the product line evolution where former variations became part of the standard package over time.

Figure 5. Conceptual Variation Model across Multiple Core Assets
The variation model shown in Figure 5 only includes the basic information. Dependent on the organization’s needs, variations can have additional properties assigned, allowing for supplementary analyses. At the end, we have a system as shown in Figure 6 that can be updated periodically with little effort. Analysis about the use of variations and therefore the health of the product line can be done continuously.

5. Conclusion and future work

The approach described here is in its early stages. We are sharing it to encourage further input from the research and practitioner communities. We plan to test this approach in a real product line as soon as an appropriate opportunity arises and will report the results.

Nevertheless, so far the approach has shown potential to be very valuable for managing variations. In particular, the ability to detect “retired” variations helps to limit their numbers.

The extraction of variation information works, but several issues must be addressed. Detection of the interrelations between variable parts in core assets is performed by matching names used for variables and inconsistent naming yields in an incomplete variation model. As mentioned before, a consistent mapping can only be achieved if all groups within the organization use the names of essential variations consistently. At the very least, the variation model allows the detection of those inconsistencies.

The graphical representations are mostly unusable because of the large quantity of artifacts. Other means to analyze the variation information must be found. A database-centric approach looks promising because database tools are built for dealing with huge amounts of data.

We did not try to use the variation management system recursively. It is conceivable that one or more groups within the organization would use the system to manage their own, local variations.

6. References


