Variability Points

What is a variability point? To answer this question it is first necessary to understand what is not a variability point, and how they arise.

Consider a software system that has, let us say, a numerical input value, two buttons and a stored value in memory. We will say that one of the buttons serves to display this value on a terminal, while the other button serves to execute an operation upon that value.

Upon startup, this system will receive a numerical value as input, possibly as a startup parameter in a terminal screen, possibly as a file that is read when it initializes. This value is stored in memory, and kept for further use. Upon clicking the Display button, this same value is now taken from memory and printed into a terminal line. The other button will take this value, and execute whatever operation is built into the system.

This is, essentially, the description of a very simple calculator, one that can only do one thing. From the programmer's point of view, no matter how many different calculators are programmed with however many different operations, the form by which the input value is read and the form by which the memory value is displayed, will remain the same. These are, then, the "commonalities" of the several calculator systems that can be built from this specification.

The variability point, then, is the one part of the system that must be programmed differently for each desired product: the Operation button. In this example it is still difficult to visualize the concepts of a Software Product Line, but it is already evident that if one considers many different systems built from this specification, each one with a different Operation button, that the rest of the system can be reused every time a new Operation is desired. To summarize, the system has three "features": the Display Button, the Input Reader, and the Operation Button, though only this last one is a variability point.

Consider now a new system, one where instead of a numerical value, a text string is used as input. The Operation Button continues to serve as a variability point, even though the Operations will be String operations, rather than Mathematical ones. However here, the Input Reader must also change in order to recognize text, rather than only numbers. While each of these two system families - Numerical Calculator and String Operator - has only the Operation Button as a variability point within themselves, the two of them together have a second variability point: the Input Reader. Where before we could maintain a single code, where perhaps only one method was changed per product, now we must consider components to be swapped in and out according to what type of system we are building.

Let us further use this example by considering that, regardless of which system we are using of the two we have detailed, we desire to have different forms of Display, or Output. Perhaps we now want the Display Button to also print the stored value to a window's textbox, or even for the value to be sent straight to a physical printer to be put

to paper. This, then, is a system where all three features have become variability points, and numerous different combinations of the three features can be resolved.

Why, however, do we consider these systems that execute possibly unrelated tasks - operating on strings, calculating on numbers, printing into a terminal, sending tasks to a physical printer - a "family of systems"? It is because of how we designed the variability points.

Every possible resolution of this "family of systems" is derived from a common pool of three basic, "abstract" functions: Read Input, Display, Operate on Memory. Though these three functions, or features, can be resolved in a myriad different ways, these "concrete" features will always take the place of one of the three "abstract" features. This means that if only of the three abstract features must be resolved differently to create the desired new system, there is only a need to create a single new component, while the other two can be reused from previous implementations of concrete features.

The importance of variability points is, therefore, not inherently in the variation of systems being built but rather in the several commonalities present between them. So long as these systems share a common set of features, a Software Product Line can be built with a common pool of components to resolve them.

SPL Granularity

Knowing what a Variability Point is, and what its importance is, it is now important to consider the concept of system granularity. In the previous example, though there be an infinite number of resolutions for the three abstract features, it is important to identify that these three abstract features are only one representation of the variability in the systems.

Consider that, building on the example we have used thus far, we now wish to implement Display buttons that print into the terminal, as before, but each one with a different font. Is it truly necessary to define a new Concrete Feature for each and every font, and implement a new Display Button for each of these features? Using the SPL Architecture we have built thus far, the answer is yes - nothing can be reused of the Terminal printing methods from other Display Buttons, because the font that they use is a part of the Concrete Feature represented in that implementation.

However, consider a different SPL Architecture where what we have so far called the Display Feature, is now divided into two different abstract features: "Output Device" and "Output Font". Now, we have created an architecture that allows for the reuse of a single Output Device - say, the Terminal - with several different fonts. What is being "switched" in the system now is no longer the Display Button as a whole, but rather a "component" of it.

What we have just done is "refine" the granularity of our architecture. We have taken one larger feature and transformed it into smaller ones, allowing us to

componentize smaller parts of this feature for reuse. The "granularity" of a system is, then, defined as "coarser" or "finer" in accordance to how refined the features are into their most basic functions.

What is more advantageous then, a finer or coarser granularity? The answer is, it depends. In the systems family we had been using in the previous section, it would have made no sense to use a finer granularity to represent font, given that the font being used in the output was the same in every single variant of the family of systems. However, in this new example we have built, the finer architecture gives us the opportunity to reuse a larger part of the code in our variants, reducing the amount of programming required for the creation of each new product.

The choice of how much to refine the granularity of an SPL Architecture therefore depends on the variants that are expected of it. It is, of course, impossible to predict each and every variant of an SPL, and therefore it is impossible to predict the best degree of granularity to use. Designing finer architectures from the beginning may seem tempting in a long-term view, but the finer an architecture is, the greater the effort will be to implement the "variability mechanisms" required to resolve it.

Variability Mechanisms

How, then, does one actually implement these variability points into code? Is it simply a matter of commenting code out? Switching methods before compiling? It can be, but these solutions can be costly.

As the number of variations for each feature grows, the amount of effort required to manually select variants for each feature becomes greater. In a small SPL, such as the one detailed in our example, it might seem tempting to simply implement a different method for each feature, and just change which method is being called on "main" before compiling. In a more practical example, this becomes unfeasable.

Consider the Linux architecture, which has been mapped by the University of Waterloo into over four thousand features. The amount of effort that would be required to manually select the code for each of these four thousand parts would be colossal - if possible at all. It might be that certain features "share code" and turning off the code for one of them might cause another to malfunction (note that this is regardless of the Feature Model stating these two features to be unrelated - the programmers might still have found a way to couple them at implementation level).

For this reason, a practical SPL must implement variation resolution mechanisms capable of automatically selected the code required to build a certain specification. These variation mechanisms can take several shapes, such as compile-time variables like #IFDEF, interfaces that are implemented by different classes to represent different features, and many other proposed forms.

The issue our research team sees with the majority of these variation mechanisms is that they become more difficult to program as a project progresses. More

often than not, the same compile-time variable will be used in multiple locations and will be difficult to keep track of, or certain interfaces might have methods begin called in different parts of the code, each one with a different purpose.

To facilitate the evolution and maintenance of an SPL code, our research has proposed the use of Factory classes to resolve variability. The framework for this, at this time, is extremely rigid, and is as follows:

Given a particular Abstract Feature, a single Factory component - restricted to a single library or dll - is created to represent it. This Factory component contains a single class, with a method that is essentially a "switch": it is in charge of choosing, based on whatever many conditions be present, which variant must be used for a particular situation of the system. When the system, during execution, comes to a point where that Abstract Feature must be resolved, the Factory is called upon to do so. The system, essentially, has access only to a "skeleton feature" containing the signature of the Abstract Feature - literally an abstract representation of the feature - and the Factory is the only one that truly has any connection to the concrete features.

The implications of this solution, as we have been able to identify so far, are primarily that the entire variability mechanisms of the SPL are contained within single packages, libraries, dlls, or whatever encapsulation mechanism be used. That way, the resolution of an Abstract Feature becomes dependent on a single point, and is no longer spread across the entire code. The programmer is effectively forced to make the system as loosely coupled as possible.

On the other hand, this solution mechanism implies that for very small features, very small libraries will have to be created. In the same example of the Linux architecture, it would mean that a minimum of four thousand libraries would need to be called upon during the execution of the system, and tens of thousands of libraries would exist to cover every possible variant. Many of these libraries would end up being made of, perhaps, one or two lines of code, depending on the granularity of the architecture.

We are bound, therefore, to a coarser SPL architecture to be able to contain variability as we want to small, identifiable pieces of code. This is one of the challenges we face: how to hold our control over the variability mechanisms, but still be able to produce finer architectures.

Shared Structures

Of special interest to our particular approach to SPL Architecture are those features that represent data structures of the system. These data structures are necessary to enable the communication between different components of the system, given that the concrete features have no knowledge of one another and therefore cannot make use of one another's internal structures. The input and output of every feature must be yet another feature, a data structure.

These data structures share similarities and differences with the features that represent functions of the system. Like them, they can also be represented by Abstract and Concrete Features, where the Abstract Feature defines what type of data structure is needed, and the Concrete Feature defines the exact details of that data structure. An example that can be used is, say, a variability point where the Abstract Feature is "List" and the Concrete Features are "Array List", "Vector", "Linked List", etc. These data structures have similar purposes and similar operations, but entirely different implementations and performances.

Unlike the "function features", however, the data structure features must in some degree be known by other concrete features to enable their communication. It is may be insufficient for a Parser to know it is receiving a Class Diagram if it does not know what type of notation is being used in that diagram; it will not know what to look for, what to parse.

For that reason, Data Structure Features have a slightly different variability mechanism in our approach. While they are still represented by "skeleton components" and resolved by use of Factory components, they are, in fact, accessible by the system's functions. The system architecture must contain a static piece of code that serves as an index of every concrete data structure available to the system, and every data structure that is instantiated must be identified by an entry from this index.

What happens is, a component that implements a "function" concrete feature will generally be aware of what types of concrete data structures it is capable of handling, and will therefore have direct access to the libraries of these concrete data structures. It will, however, continue to use the factories to identify the data structures at the time of input and output, only casting them into a concrete structure during its internal use.

Consider our earlier example of the Display and Operation buttons. Consider that a particular operation can be executed upon every single numerical type available. The function component for this particular feature will have no need to identify a concrete data structure; it merely needs to request that the Factory identify its input and output as a Numerical data structure, and that will suffice. Using the operations made available in the skeleton component, tipically an interface, it will be able to execute its code without ever having access to how this code was executed. Likewise, any operation that can be executed upon any Text format will only require the factory to provide it with a Text data structure.

Consider now, however, a particular operation feature that can only be executed upon floating point numerical types: float and double, specifically. This operation is aware that the data structures it is able to handle are float and double, and therefore has direct access to them. It will still receive as input an object of type "Numerical", and will still cast its results into an object of type "Numerical" for output. However, before beginning its operations, it will seek out the part of the data structure that identifies it in the index, will match it against the static index component, and will cast it into the

appropriate concrete structure - either float or double - or launch an exception stating that it cannot operate, or does not recognize, that particular concrete structure.

In order to make such an index available, a new concept of SPL Architecture arises that is important to discuss: that of the SPL Core.

SPL Core

Aside from all the features that are identified in a system, be they variability points or not, there are certain control aspects of a system that will be required in absolutely every variant of that family, parts of code that are mandatory no matter what product you wish to make. The data structure index is one of these aspects, but there may be others.

Often, the Core in our approach is comprised of just a couple of things: the data structure index, and the system initializer. In any system domain that has any concept of process or linearity, there will be the need for a central controller that seeks to accomplish this process by use of the components available to it. This controller is responsible for, at the very least, booting up the system and picking out a possible central control feature, but often also involves the management of certain static data structures, the input and output of data, the communication with the operating system and other basic tasks.

It is important to remember that the Core is *not* a feature of the system; it cannot be reduced to an abstraction and cannot be used as a variability point. The Core of the system is comprised only of those parts of the system that are *mandatory* regardless of the variants selected. Therefore, a minimalistic approach to the Core is advised, as decoupling features from it has proven to be a difficult task in our past experiences.