Summary: Managing product variants in a component-based system

Jacob Gradén, Anna Ståhl
Lund University, Sweden

{*graden*, *aannastahl*}@gmail.com

Abstract

Today's markets are fast-paced, with many different customers and requirements on products. Component-based systems are becoming a popular way of managing product variants and promoting code reuse. The many different requirements and product variants introduce complexity which needs to be managed while retaining flexibility, so that creating product variants is facilitated. This means that not only components and products must be managed, but also information pertaining to them, such as technical relationships between components and business requirements on products.

This master thesis suggests a support tool to help in creating and managing the different components and products, and outlines the capabilities such a tool should have and the opportunities it would present.

Keywords: Component-based system, Composition, Configuration, Configurator, Dependency, Rules, Variants

1. Problem area

At its core, the central problem is simply this: *How can the variants of a component-based system, and their constituent components, be managed efficiently?* This is however a large field, with many facets.

A component-based system can be very flexible. Components are meant to be mainly independent, which means they can be combined freely to create any number of variants. There may however be relationships between components – for example when one component needs another to function properly, or when two components are mutually exclusive.

The resulting combination of components (the *configuration*) may also impose restrictions on possible combinations. For example, components may belong to different layers of the system – such as operating systems and applications – and one restriction might be that at least one component from each system layer is present. Components also have properties, and another restriction might be that all components in a configuration have the same value for a specific property.

There is a lot of complexity involved as well. The sheer number of components in a large CBS makes it difficult to manage manually, and when components are developed and exist in different versions over time, the amount of components becomes staggering. Add to that the possibility to create configurations, and variants thereof, by combining components in different ways, and it is clear that complexity will become a major hurdle.

Using CBSs allows for great flexibility in creating variants of a product, facilitates parallel development and can reduce compilation times. Using a support tool for managing CBSs keeps the time and costs down, simplifies communication between departments, allows for statistical analysis of components and product variants, and greatly reduces the risk of creating broken or nonsensical product variants. Such a tool could also mitigate the risk of manual errors, provide statistics of component usage, and enable the tracking of license costs, among other things.

Figure 1 illustrates an overview of the composition process.

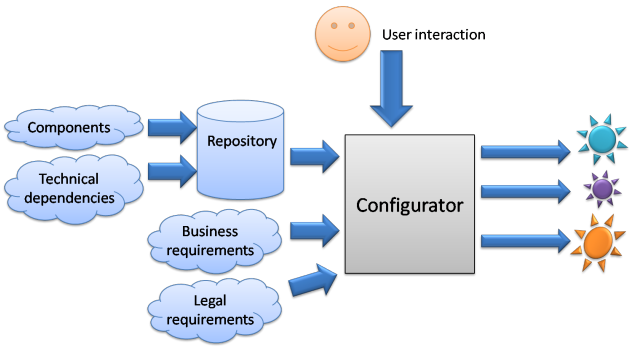

Figure 1. The composition process

2. Findings

Using component-based systems allows for great flexibility, but at the cost of complexity. That complexity needs to be handled, and a semi-automated tool for helping to do so would be very valuable.

Components Components generally need to be *independent* of each other, and when they are not, the relationships need to be *explicitly documented*. Components have many *properties*, notably *market*, *customer* and *test status*. The test status may for instance take the values untested or approved, but many more can be envisioned.

Relationships Components may have *many different types* of relationships; among them requires, conflicts and replaces. Components may also have indirect relationships in the form of *feature dependencies*, which allow a loose coupling between service-providing and service-using components, or *component suites*, which is a method to group components into suites. Relationships are purely *technical*.

Rules Rules correspond to a higher *level of abstraction* than relationships and can be used to define requirements between components, component relationships, component properties, etcetera. They allow for managing more advanced associations between these items, thereby lessening the manual labor, and can be limited in time or space. Rules can be *explicit* or *implicit*, and different *types* of rules may be necessary, such as *business rules* or *legal rules*.

Configurations A configuration is created by specifying which *components* and their *versions* that are needed. Configurations have properties of their own, such as *production stage*, but also *receive property values* from their constituent components – an untested component, for example, means the entire configuration can receive a test status no higher than untested. Configurations may *vary greatly in size*, from single components sold as separate applications, to complete software systems.

Repository The repository stores all *configuration items* and their *metadata*, such as relationships, test status, etcetera. Changes and deletions are generally not allowed, only additions; but for some metadata, other considerations apply. To keep the repository clean, *commit checks* and controls for modifying items are needed.

3. Support tool sketch

Figure 2 and Figure 3 exemplify what a support tool could look like. All components are visible at all times, and selected, required and unselectable components are clearly marked. Test status and version information is also available to support the user. Every time the selection of components is changed, the list of components is updated. This real-time update allows the user to easily see the impact of changes.

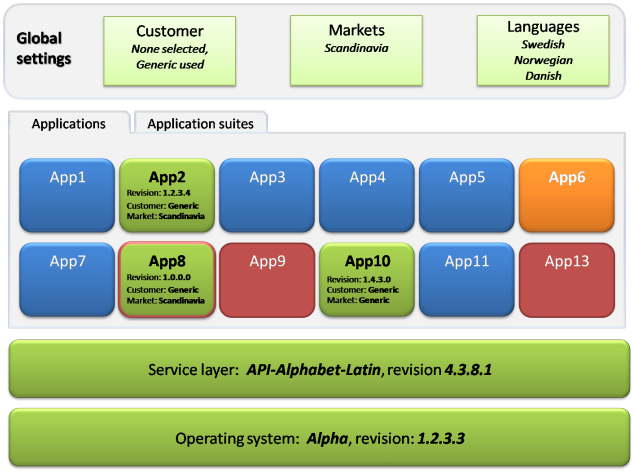

Figure 2. Support tool, example 1

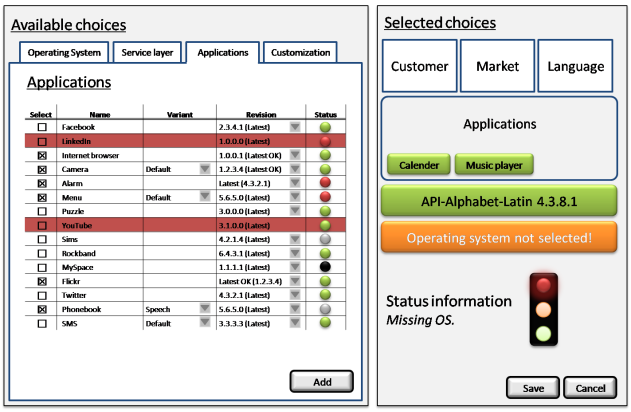

Figure 3. Support tool, example 2

Complete details are available in the full report, *Managing product variants in a component-based system*.