

BCS CMSG Seminar
Why Software Asset Management and Configuration Management is essential
London, March 17th 2004

Flexible Configuration Management for a Component-based Software Asset Repository

Tom Brett
Consultant, Confluence Systems Ltd
in association with MKS

Abstract

The paper describes techniques for managing a repository of multiple re-usable software components. Business requirements, such as customer-specific product variants, may demand multiple concurrent product configurations assembled from the same software asset base. Parallel development potentially needs to be managed at both the component and product level, recognising that different products and components may have their own independent development life cycles.

This paper draws on real-life experiences to illustrate how to achieve the necessary flexibility while maintaining an all-important degree of control over diverse configurations. MKS Source Integrity Enterprise is used as an example of a modern, enterprise-capable SCM tool which may be used to implement such a repository.

Copyright © Tom Brett 2004
MKS Trade Marks acknowledged

Why Re-usable Components?

This paper discusses the Configuration Management issues, challenges and solutions arising from a repository of re-usable software components. Note that there is no implication that re-usable components are a “best practice” which will suit every software development organisation. Rather, this paper begins from the premise that a component-based pattern has been found to be appropriate, and deals with the specific Configuration Management issues.

Motivation for re-usable components may include the following:

- Multiple development teams using a largely common software infrastructure: sharing common code libraries between teams can avoid “reinventing the wheel” and improve overall efficiency.
- A modular product strategy based on a component library (frequently found in the mobile devices industry), where each product is just some different combination drawn from the component “palette”.
- The ability to tailor a core product for different customers, by custom modifications and additions.
- Note that these requirements may be combined or nested; e.g. a core product may itself be modular in structure

Component-Based SCM

In order to support a component-based pattern, an SCM tool must provide certain key functional elements:

- Version control at the individual file (or *member*) level.
- Grouping of files into component (or *subproject*) configurations, and the ability to version-control the entire configuration – i.e. which specific member versions make up a version of the sub-project.
- Re-usable grouping of components into product (or *project*) configurations.
- Many-to-many relationship, i.e. a project consists of many components and a component may be used in many projects.
- Arbitrary nesting depth of projects and subprojects.
- Very importantly – the capability for an independent release cycle, version and branching history for each component.

Introducing CM "Patterns"

Most readers will be familiar with the well-established concept of "Patterns" in software engineering. A pattern is an established engineering solution for a certain class of problems: for example, in civil engineering, the "river crossing" class of problems may lend itself to solution patterns such as "bridge", "ferry" or "tunnel". The engineer will be guided to a choice of pattern by parameters such as width, depth and flow speed of the river, type and density of the expected traffic, and so on. In both choosing a pattern and subsequent execution of the solution, the engineer can draw on a body of prior experience with these patterns.

The use of patterns is less mature in Configuration Management than in software systems analysis and design, but is still a very useful and instructive approach.

This paper will now propose two patterns which can be applied in different circumstances to achieve a re-usable component-based CM repository. Again it is worth noting that these patterns may – indeed, should - be adapted and/or combined (with each other or with other CM patterns) to achieve the best solution fit with a particular set of development project constraints and requirements.

“Component Palette” Pattern

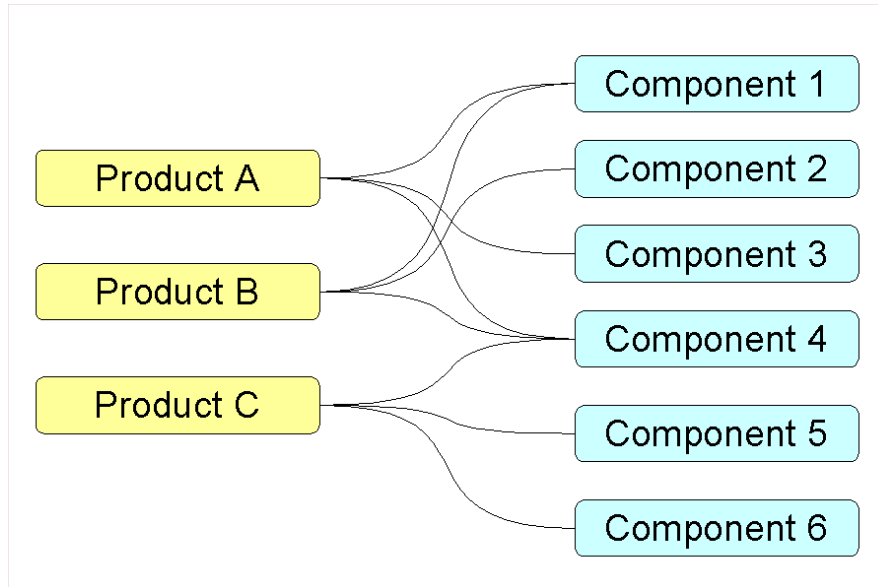


Illustration 1 Component Palette Pattern - conceptual diagram

The “Component Palette” is conceptually a very simple many-to-many relationship between a flat-structured set of products and a flat-structured set of components.

Characteristics of the Component Palette:

- Flat structure
- All components are treated symmetrically – even if they are only included in one product
- Symmetrical treatment makes it easy to re-use code later, even if not originally planned to do so.
- “Large library – small product”
- Each product typically includes a relatively small percentage, say 25%-50%, of the whole code base.
- (Often but not exclusively) the release history is concentrated at the component level, while products may have little release history.
- Different products may be linked to different versions of the same component(s).

A good example of a situation which lends itself to the “Component Palette” pattern is software development for the mobile devices industry. Here, individual components may have an extensive incremental development history, including variants for different hardware types. Products, on the other hand, may have a very short release cycle due to the need to “get it right first time” before a device goes on the market. Development is then likely to move on to new products rather than to revised versions of existing ones.

The mobile devices scenario also illustrates the need for decoupling the development and release cycles of the components from those of the products. It is vitally important to allow products to reference tested and validated versions of the components, while not preventing new development from proceeding on the “tip” of a component's history.

Tool Support for “Component Palette” Pattern

Support for the Component Palette in MKS Source Integrity relies on using a combination of two key features:

- Development Path (defines branching at the project or subproject level)
- Shared Subproject (allows a project to be virtually included in a higher level parent project)

The Shared Subproject is analogous to creating a “link” in a Unix file system: while a Unix link makes the contents of a certain directory accessible through a different location in the file system, a Shared Subproject creates a link in the Configuration Management repository to allow a subproject to be logically included in a parent project, irrespective of its physical location. Because it is not physically included, it is possible for the same subproject to be shared by multiple parent projects, satisfying the “many-to-many” requirement of the pattern.

Furthermore, it is possible to link different parent projects to different versions or branches of the same component, satisfying the “independent release cycles” requirement of the pattern.

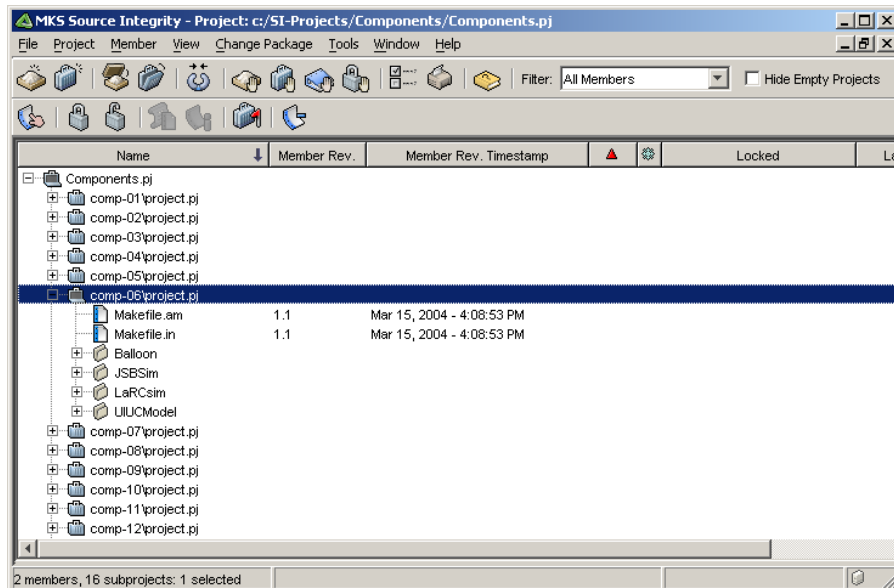


Illustration 2 Component Palette in Source Integrity

In Illustration 2, the Component Palette has been implemented in the Source Integrity repository as a set of (physically contained, not shared) subprojects under a “Components” parent. This is done purely for convenience of grouping, and is not mandatory. One component has been expanded to show its internal member and folder structure.

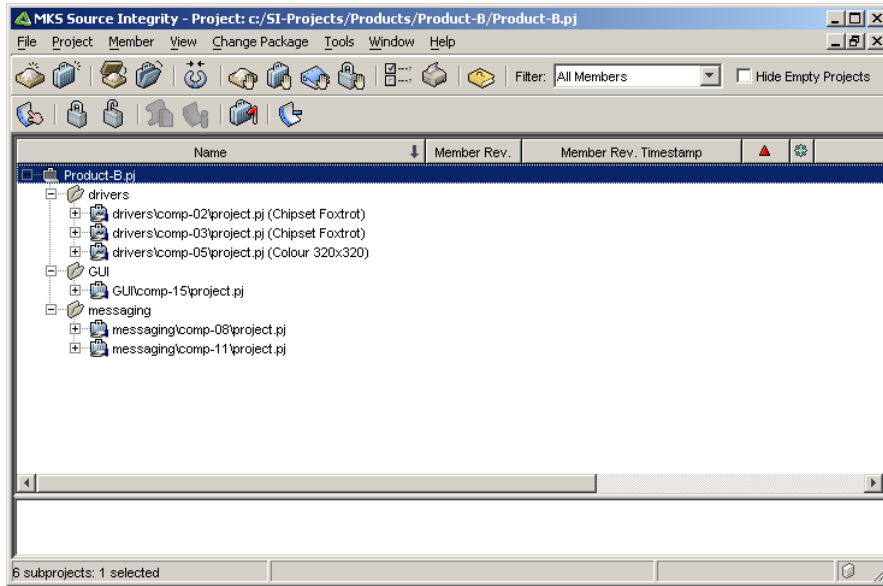


Illustration 3 Product using Components in Source Integrity

In Illustration 3, a Product has been implemented to include the required Components by means of “shared subprojects”. Note that (just as in the Unix links analogy mentioned earlier), the subprojects may be shared into any desired directory structure under the parent project. In this example, although the physical component repository has a flat structure (Illustration 2), the Product-B parent project shares the subprojects into three different subdirectories, “drivers”, “GUI” and “messaging”.

Note also that for the three subprojects under “drivers”, the subproject share has been linked to a particular development path (or branch) within the target component subproject. Illustrations 4 and 5 show the branched project history for two of these components and also emphasise once again that each component may (and, in general, will) have its own independent development and release cycle.

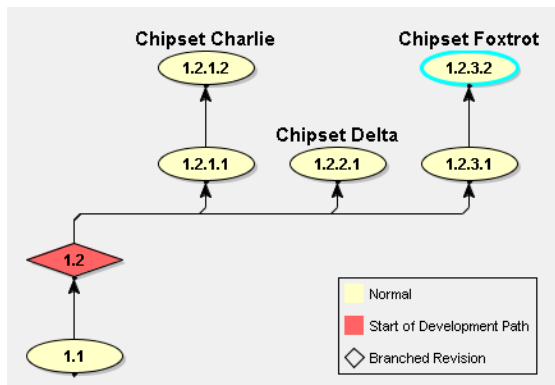


Illustration 5 Project History, comp-02 component

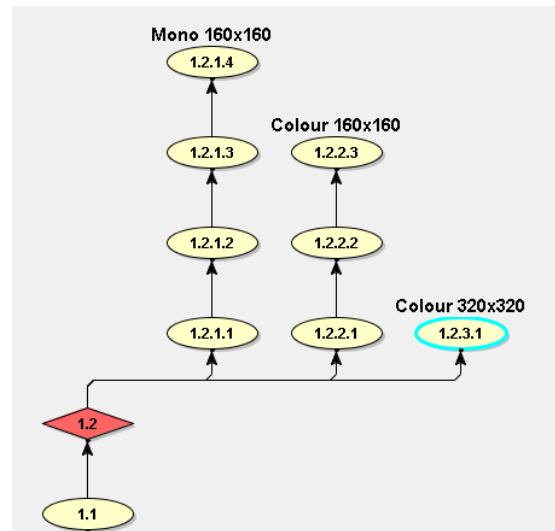


Illustration 4 Project History, comp-05 component

“Tailored Core” Pattern

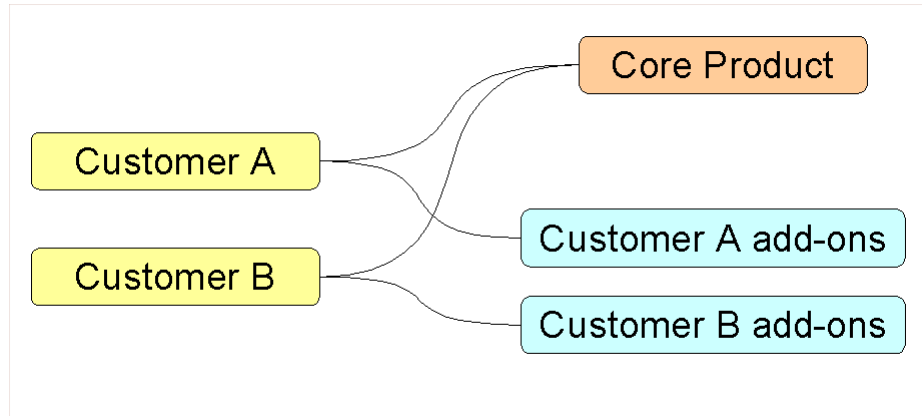


Illustration 6 Tailored Core Pattern - conceptual diagram

The “Tailored Core” is a somewhat more structured pattern than the “Component Palette”. One component is designated as the “Core Product” and is referenced by all the releasable products. Typically, as shown in the diagram, releasable products are targeted at different customers and will include customer-specific components in addition to the Core Product.

Characteristics of the Tailored Core:

- Core Product usually contains complex sub-structure.
- Core Product code is typically not re-used anywhere else.
- Core Product frequently contains customer-specific development branches.
- Customer-specific add-ons are implemented by re-usable components.
- “Large product – small library”.
- Each customer release typically includes large percentage, say 75%-85%, of the whole code base, with a large degree of commonality between customers.
- There is often a long and complex release history for each customer, as well as a complex history of the Core Product itself.
- Very often there is only a small number of customers, and the software supplier has an intensive relationship with each one.

Instances of this pattern would typically be found in applications such as industrial automation and management information systems, where software vendors frequently provide tailored solutions based on a core product.

Tool Support for "Tailored Core" Pattern

Support for the Tailored Core in MKS Source Integrity utilises the same key features as for the Component Palette (Development Path and Shared Subproject) but with a somewhat different structural emphasis.

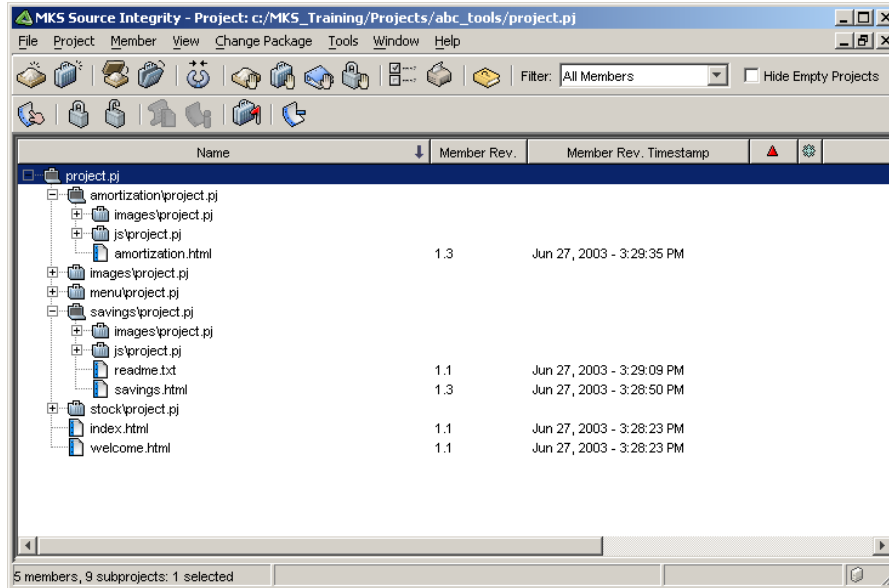


Illustration 7 Core Product in Source Integrity

Illustration 7 shows a typical source code project with a relatively complex internal structure, including a number of (physically included) subprojects of its own, which will be used as the core product in this example. This project has its own branched history (Illustration 8) showing development paths for two core software releases and a further, customer specific, variant branched off from Release 2.

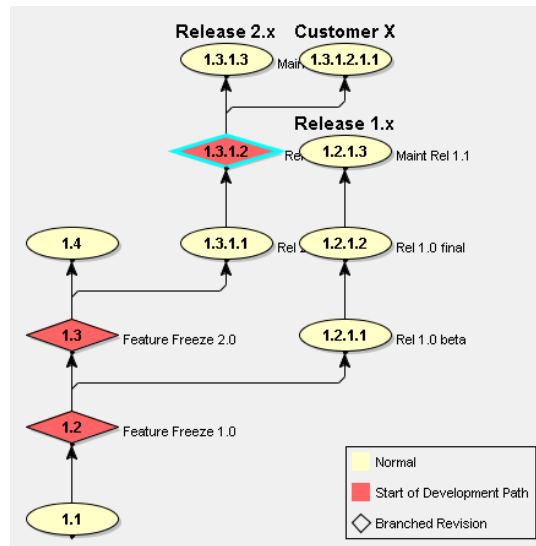


Illustration 8 Core Product - Project History example

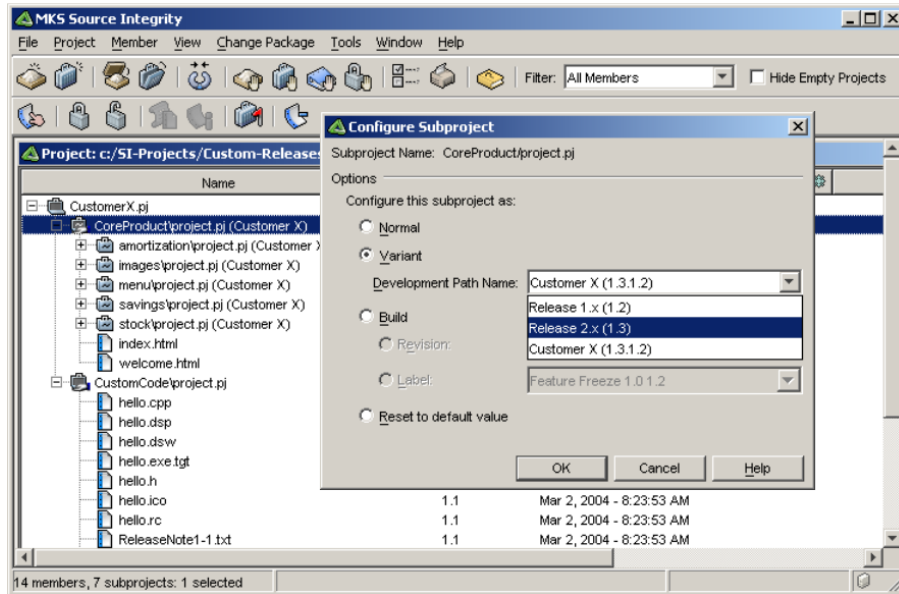


Illustration 9 Configuration of Core Product and Additional Component(s)

Finally, Illustration 9 shows how the customer-specific variant of the Core Product is assembled into a release together with an additional component specific to that customer. This is achieved by sharing both as subprojects within a “container” project. The illustration also shows how the subproject can very simply be re-configured to point to a different variant if requirements should change: in the example, the configuration is being changed to the basic “Release 2.x” variant instead of the customer-specific one.

Conclusions

This paper has attempted briefly to show how component-based Configuration Management patterns, implemented using a capable Enterprise SCM tool, can make it possible to manage the re-use of software assets within a development organisation.

The key benefits of this approach are:

- **Responsiveness** to fulfil customer needs by combining standardised and customised components
- **Re-usability** to maximise efficiency by avoiding duplicated effort
- **Flexibility** to easily match configurations to requirements
- ... resulting in a **Competitive Advantage** for the organisation employing this approach

About the Author

Tom Brett has been a software development professional for 20 years (starting with scientific Fortran programs and later moving on to C and C++) and gave his first conference paper on Configuration Management as long ago as 1992!

Tom has worked for clients such as Citibank, Siemens, British Telecom and 3Com and is now back in independent practice after a spell of 3 years employed as a consultant by MKS.

e-mail: tom@confluence.clara.co.uk

