Strategic reuse and product line engineering

Using the IBM Rational systems and software engineering platform

Eran Gery
Joanne L. Scouler

Introduction

With the growing complexity of many products and intensifying competitive pressures, the need for engineering development efficiencies is increasing. This is true across many industries, such as air, land, or sea vehicles, medical devices, and consumer electronic devices, to name a few. In all of these industries, delivery of multiple products sharing common elements is standard practice, resulting in product lines or product families. Reuse of assets across a product line or family is a major efficiency improvement for easing product design pressures.

There are many ways to reuse assets; some ways may not be as efficient as others, but it makes sense to utilize existing work. When a company reapplies assets and technology to new projects, a subsequent change to an asset in one place is often not propagated cleanly and efficiently in all the places where the asset is used. Duplication of work is likely if there are not processes and tools in place to automate reuse.

Developing an orchestrated process at a high level in the company for maximizing reuse of assets across the product lifecycle is important for increasing design efficiencies and taming complexity. Strategic reuse is a high-level business strategy for reuse of assets.

This article explains how to do strategic reuse within product lines.

The solution proposed in this article solves these key problems:

• Wasted time and money developing components that already exist in other company products
• Inefficient propagation of changes to assets because of difficulty in knowing where assets have been reused
• Compromised product quality due to ineffective, error-prone manual processes

The Rational solution for systems and software engineering

The Rational solution for systems and software engineering (SSE) platform, is built on the Jazz™ platform and the Open Services for Lifecycle Collaboration (OSLC) protocols. OSLC facilitates integration between different lifecycle tools and repositories such as requirement management tools, design and analysis tools, implementation, and test management tools.

Reuse scenarios in complex systems development

A common scenario in a complex systems development project is shown in Figure 1. The scenario shows a base product that is being developed over time, with variants also being developed. Variant 1 is developed first, followed by variants 2 and 3. The variants share considerable commonality, and they are based on a common platform. The common platform is realized as a collection of engineering assets, and it includes requirements, analysis and architecture models, implementation artifacts (such as source code), and testing artifacts.

Clearly, the goal is to effectively reuse the platform engineering assets across the variants, with proper management of commonality and variability. It is important to be able to manage artifact sharing and allow for the creation of artifact variation. While the platform artifacts are being developed (represented by the “core platform” arrow), they go through a set of changes (such as defect fixing). The changes are propagated from the platforms to the variants as shown by the dotted arrows. A change often originates in one of the variants and later needs to be propagated back to the platform development team, so that other variant development teams can use it. The key to managing strategic reuse is enabling parallel development of the core platform and multiple variants while providing accurate upstream and downstream change propagation.

Figure 1. Developing a family of products
From ad-hoc reuse to strategic reuse

There are many ways to reuse engineering assets. One type of reuse is commonly known as "clone and own." Assets are copied and then maintained by the new owner. As changes are made in the original or the new copy, they are not easily propagated. The changes are not propagated because you do not know how many times a clone-and-own operation occurred and where the cloned assets are located. The clone-and-own approach does not scale well, so more efficient means are needed to manage and promote reuse of common assets across products in a product line.

Another pattern of reuse is defining a product configuration across multiple asset types, which are referred to as components. The product configuration structure facilitates component reuse in multiple products. You create and manage product configurations across all your tools: requirements, tests, source code, and other assets. You can create new product variants by copying an existing product configuration and changing it by replacing or removing certain sub-configurations in its hierarchy.

The following levels for reuse, or reuse patterns, describe efficient ways to reuse assets.

Fundamental levels of strategic reuse

There are different ways to design an engineering architecture for reuse. These levels are not exclusive to one another; they can be mixed and matched and customized for particular needs. Product line development involves three architectural levels:

- **Level 1**: The fundamental level provides the core mechanism to manage product variants across streams. Configuration management concepts similar to those in source control management (SCM) systems are applied across all lifecycle disciplines, based on a federated configuration management framework. This capability leverages artifact versions, component configurations, and hierarchical baselines, as explained in the Multi-stream variant management section of this article. This level is the foundation for all strategic reuse capabilities in the Rational SSE platform.

- **Level 2**: The second level adds parameterized assets. A single source of the artifact is maintained rather than separate manual streams. The artifact contains parameters that are instantiated based on parameter configurations. Parameterized assets consist of optional elements and property values for the parameters (see Multi-stream variant management), which provide the means to instantiate the asset. Parameter configurations define how to instantiate variants from a product line.

- **Level 3**: The third capability applies feature models. Feature models abstract the variability of the system for external stakeholders, and specify variants based on feature profiles. Features are mapped to parameters so a particular feature profile can drive the feature configurations that can instantiate variants following the capability of level 2. Feature models can be done top-down, so parameters are driven from features, or bottom-up, so features are mapped to existing parameters. Feature modeling is done by interfacing with 3rd party feature modeling tools that integrate with the Rational SSE platform (see Parameterized reuse).
Multi-stream variant management

Product variants share a high degree of commonality, but you must also manage the variability across them. The primary way to manage variation is with artifact versions in configuration management.

The first architectural layer provides a branching mechanism to create new variants from previously defined products or variants, as shown in Figure 2. Each branch, called a stream, manages the artifact versions for a variant. In cases where a common platform is maintained, there will also be a stream that manages the common platform artifacts.

Each product variant is a collection of lifecycle artifacts – requirements, design models, code, tests, and so on. Each variant progresses in time through a succession of baselines, shown as circles. Each baseline records a particular state for the artifacts in the variant. When a child variant is branched from a parent variant, the child typically starts from a baseline of the parent and then continues with its own timeline of baselines. The slanted lines represent branching into variant streams.

Figure 2. Multi-stream variant management. Each variant is a collection of artifacts from various tools.

![Multi-stream variant management diagram](image)

This multi-stream variant management framework is at the core of the strategic reuse support in the IBM Rational SSE platform. There are two types of variation; in both, artifacts are managed as shown in Figure 2:

- Functional variation, which represents a different set of capabilities
- Temporal variation, which represents different releases of the same product over time

Streams are used to manage the evolution of artifacts over time for a certain functional purpose. If you develop different functionality in parallel, such as two variants of a product, you use different streams. Each stream manages the evolution of the artifacts over time.

In general, streams maintain insulation, so changes propagate into artifacts only intentionally. The changes delivered by engineers into a stream are logically grouped as change sets. Change sets can also be delivered across streams, especially for changes to common artifacts. By delivering change sets commonality can be maintained across streams. The multi-stream system provides a controlled way to maintain commonality in artifacts in the presence of change. Change set
delivery can be done in different levels of automation. The changes are tracked as part of an overall change management process that is associated with each stream.

**Components and configurations**

Complex systems consist of a large set of artifacts, which are usually partitioned into logical units called *components*. Components are important strategic reuse building blocks. Components enable better partitioning of work and supply chain processes. Component reuse for engineering and software artifacts is based on similar methods that have been used in the physical and mechanical domains for many years.

Each lifecycle tool contributes a particular collection of artifacts. For example, you might have a component of requirements, and a component of modeling artifacts. Components from several tools might be grouped into a *lifecycle component*. So for example, an engine lifecycle component might contain requirements, model artifacts, code, and tests for the engine.

Each circle in **Figure 2** represents a set of component baselines that make up a baseline for the entire product variant, which is shown as a box. A *baseline* is a special case of the more general concept of *configuration*. A configuration of a component selects a particular *version* for each of its constituent artifacts. A baseline is a configuration that cannot be changed, because it freezes the selections of a changeable configuration at a particular time. In the SSE platform components are the units of reuse, and component configurations are the building blocks of all systems.

**Figure 3** illustrates multiple configurations of a component in one lifecycle discipline, in this case a requirements project for a power train. The product has a baseline configuration. Then slightly different configurations are developed to meet the regulatory and specific needs of three different markets. The power train requirements component contains five requirements, A1 to A5. Requirement A5 has three versions, A5.1, A5.2, and A5.3. Two versions of the requirement could differ in the values of their properties: one cylinder requirement might have a 3" diameter version and a 4" diameter version.

**Figure 3** shows four concrete configurations of the power train component.

- The base configuration specifies a set of requirements to be reused by the car models. Configurations for model A, model B, and model C represent requirements that are specific to the respective car models for three different markets. They were all branched from the base.
- Each model selects slightly different versions of the requirements, but all four configurations share the same versions of requirements A1, A2, and A3.
- Note that the actual requirement artifacts are shared across the different configurations; they are not copies.
- Requirements A4 and A5 have different versions across the four configurations, reflecting that those requirements vary across the product line.
- Requirements A6 and A7 are selected only by the model C configuration. Configurations not only select versions of artifacts; they also select which artifacts are excluded or included. Artifacts can be completely excluded from certain configurations.
A configuration evolves over time. At different times it might contain different sets of artifact versions. Such a configuration is also called a stream. Some configurations, called baselines, can't be changed. Baselines are snapshots of configurations (streams) at a certain time. Their role is to record a configuration state for either a comparison or simply later reuse.

Each lifecycle tool is a component and configuration provider. Also, different tools often implement configuration management differently. Nevertheless, some of the Jazz-based tools, such as Rational DOORS Next Generation, Rational Design Manager, and Rational Engineering Lifecycle Manager, share a common configuration management service. This service manages version artifacts for these components, allows the definition of component configurations, and provides a way to obtain the correct version of an artifact in a configuration.

Other tools such as the source control management (SCM) capability of IBM Rational Team Concert™ (RTC) provide configuration management of files, including source files. RTC has a rich set of capabilities for managing versioned objects and components with their configurations, and for performing diff and merge operations (also called delivery) across these configurations. Rational ClearCase® supports similar concepts, which are more explicit in the ClearCase UCM implementation.
Note that the multi-stream variant management framework allows different lifecycle tools to use their own configuration management implementations. However, these different implementations must be consistent in how they define and use components, configurations, and baselines.

**Global configurations**

Components in each tool manage different configurations to enable more fine-grained variation within the artifacts for what you are building. As mentioned earlier, products consist of multiple components, each with multiple configurations. Therefore, to manage the entire configuration of a product, SSE uses *global configurations*. A global configuration is a composite of configurations from several tools across the lifecycle. Global configurations can contain other global configurations, forming a hierarchy of configurations that typically corresponds to the hierarchical nature of a complex system. **Figure 4** illustrates a hierarchy of global configurations that represent the engineering assets of a car model.

The top configuration, Model GL, contains an entire car model configuration. It contains components from various lifecycle tools, including the requirements, overall architecture, and product level tests for that particular model of the car. It also contains other global configurations for the primary subsystems of the car: the power train (P) and the body (GL). The power train then contains the configurations of the engine (4C) and the gearbox (6S).

**Figure 4. Component composition and hierarchical baselining**

The global configuration of a component determines the configuration of its subcomponents. That is, the GL variant of the car contains the GL configuration of the body and the P configuration of the power-train. Note that all these named configurations can vary over time, similar to configurations created by one tool, such as requirement configurations.

Global configurations can also have baselines. Essentially, the product baselines in **Figure 5** are global configuration baselines.

Since global configurations can be hierarchical, their baselines can contain other baselines. This is sometimes called *hierarchical baselining*. In **Figure 5**, a baseline of the car model (2014 I1) implies baselines of the body (2014 I1) and the power train (2013 I4) configurations, and a baseline of the power train implies a baseline of the gearbox (2013 I3) and the engine (2013 I3) configurations.
Global configurations are based on an open standardized OSLC configuration management protocol. Any tool can contribute configurations as part of the global configuration if it supports the OSLC configuration management specification.

**Defining products and variants with the global configurations editor**

The global configuration editor facilitates definitions of product variants as hierarchies of lifecycle component configurations. In a tree-oriented user interface, you can specify product configurations from component configurations in a hierarchical manner. Basically, the tool defines, manages, and visualizes how product variants are made from components and their configurations. The nodes in the tree represent component configurations – internal nodes correspond to global configurations, and leaf nodes represent configurations that are provided and managed within the lifecycle tools.

**Figure 6** illustrates a car variant as defined with the global configurations editor. The product nodes represent component configurations, where internal nodes are global configurations, and the leaf nodes are associated with configurations in lifecycle tools (also known as local configurations). The top node represents the entire car product configuration. The children of the Car node are the two main subsystems of the car, Power train and Body. The Power Train variant is Basic, and the Body variant is Standard. The leaf nodes (in different colors) link to the respective requirements, code, and test configurations in the respective tools.
A key operation in the multi-stream pattern is to branch a new variant from an existing product configuration. For example the GL car is branched into a new variant, named LX. To branch, you "copy a configuration". Then you update the configurations within the structure with the components the new LX car will use. Figure 7 shows how the configurations editor visualizes the two product variants. In the LX variant, you see the new turbo variant of the engine, with the associated configurations for the requirements, code, and test cases.
Variant management practices

Once you know how to maintain artifacts and components across product variants, you can move on to the other activities or use cases involved in developing a product line:

- **Creating a new product variant**: You can base the new variant on the common platform or a more specific variant, and use each tool (for requirements, models, code, and tests) and the global configuration tool to carry out this task.
- **Baselining a variant**: After changing various components or reaching a milestone, a common practice is to set a baseline for a variant. You can assemble baselines created in each tool into a global baseline of the entire variant. Tools enable tagging of baselines to ease this process.
- **Navigating across components of a variant**: Traceability links facilitate navigation from a test case to a requirement for example. After changing a requirement you need to update the proper version of the test case. When you click the link, the system fetches the correct version of the test case based on the global configuration of the variant.
- **Propagating changes into and from a variant**: When changes are made within one product configuration, you sometimes need to propagate them to other product configurations. For
example, when you change a common requirement, you can cascade that change to all the variants to maintain commonality. The tools enable propagation of changes across configurations, such as from a common configuration to a variant and back.

• **Finding where components are used:** When you identify an issue in one variant, the tools help you find other variants that reuse that component, so fixes can propagate to all the required variants. The impact analysis of a change in one place indicates what other variants need to change.

• **Comparing variants:** Quick comparison of the structure of two variants is essential, showing both commonality and variability. The tools facilitate comparison of the component structure of two variants, and you can also see a detailed comparison of artifacts.

### Parameterized reuse

You now know how to manage variability by selecting different versions of artifacts inside configurations. The different versions can be created by branching artifacts and changing them.

More complex systems are often characterized by a very large number of variants, and to further streamline the creation of variants, you could derive variants from a common generic platform. The generic platform components contain parameterized artifacts representing a generic product line. The parameterized artifacts might contain optional elements, or parameterized attribute values. The specific values of the attributes or existence of optional elements could then be derived for each product.

In the generic platform with parameterized artifacts, different variants of the artifacts could automatically be derived by providing concrete parameter configurations, as opposed to manually branching and modifying them. As a result, complete product configurations could be automatically derived from the platform by applying different parameter configurations.

**Figure 8** shows an example of a generic product structure that contains components with conditional expressions that are based on a set of parameters. The parameters are defined in a designated parameter artifact.

The product tree shows a generic car structure (car), which contains five optional components. Engine has two optional variants, labeled with the values "Gas" and "Automatic." GearBox also has two variants, with the values "Automatic" and "Manual." They are controlled by the variability parameter $p_{GearBox}$. The third optional component in the structure is the sunroof. Each one of the component references in the structure has a conditional expression that could determine which components to include based on the setting of the corresponding parameters.
Figure 8. A generic car platform with variability parameters, managed in Rational Engineering Lifecycle Manager.

From the generic product structure specific configurations can be derived by creating a stream of the generic product and providing a set of concrete parameter values. Figure 9 shows a product line structure that is derived from the previous generic structure:

Figure 9. Derivation of a car product based on the parameter configuration

The variant structure in Figure 9 is created starting with a new product configuration node and specifying a parameter configuration as a dedicated node in the product tree. The parameter node can define specific parameter values for the variability parameters. In the example, the parameters pEngine, pGearBox, and pSunroof are set to "gas," "automatic," and "false" respectively. Eventually, those parameter values could be consumed by the conditional expressions on the product nodes (where they exist), and determine which variant node to select for that specific variant structure. The "turbo" engine variant, the "manual" GearBox variant, and the Sunroof component are excluded from the configuration shown.

As illustrated, parameterized product structures could be handled by creating variant streams just like for non-parameterized systems. However, in parameterized product structures the content of the variant might be determined partly or fully by derivation from a generic platform through parameter configurations.

Figure 10 shows a pattern in which the product line contains parameterized assets and defines a set of variability parameters. Each variant can define a parameter configuration giving values to some or all the variability parameters, hence instantiating the product line into a product. Note
that as opposed to basic branching, here the variants contain artifact versions that have been automatically derived from parameter settings.

**Figure 10. Parameterized variants**

Most variants branch from an early baseline of the product line and development proceeds in the generic product line. Baselines for variants are then derived from product line baselines by instantiating the parameterized generic artifacts according to the values supplied in the variant's parameter configuration. This derivation is indicated in Figure 10 by the dashed arrows (these arrows were used in Figure 2 to indicate propagation. Derivation is like propagation followed by instantiation).

However, nothing prevents development in product variants, resulting in some artifacts not being automatically derived from the product line. Changes can be reflected back into the product line using the standard mechanisms of change delivery.

**Feature-driven reuse**

Feature modeling can further augment variant management when combined with the previously described reuse patterns. The variability of a product line could be abstracted using feature models. Feature models represent the variability of a system as a hierarchical tree, made of nodes called "features". Features represent user visible capabilities of the system. The feature model depicts the feature variation of the system, as it presents alternative and optional features that would specify a variant in a product line. Features could be assembled into a feature model using relationships to depict mandatory, alternative, or optional features.

Alternative features imply that one of the possible features has to be selected, as shown in Figure 11 with a solid triangle grouping the incoming edges. Optional features can be attached to an edge using a hollow circle. Simple edges represent mandatory features in the context of their parent. The feature models can also show constraints among features – for example, a hybrid engine that requires a trip level of xlt. The constraints could help product line managers specify a valid combination of features. A variant is characterized by a set of features.

Technically, the set of features characterizing a variant are similar to how parameter configurations are applied to a generic product to derive variant specific artifacts. Therefore, feature models could be layered on top of parameterized components by associating features with the variability parameters of the solution. The parameters presented in Figure 8 represent similar choices to the feature choices illustrated in Figure 11.
Figure 11. Car feature model

Figure 12 shows how a feature selection of this same feature model maps to a parameter configuration that drives the configuration of a variant artifact.

Figure 12. Feature selection driving variant-specific artifacts

To derive sets of parameters from feature selection, a mapping framework might be necessary between features and the parameters used by the assets, in case there is no 1:1 correspondence between them. The mapping framework must be provided along with the feature modeling capability.

The mapping between features and variability parameters is used in the AUTOSAR specification framework for automotive systems (in AUTOSAR variability parameters are called system constants).
Feature modeling services are currently not directly provided by the Rational SSE platform. They could be made available through interfaces between the Rational SSE platform and feature modeling tools such as Gears by Big Lever Software and pure::variants by pure-systems GmbH.

**Conclusion**

The reuse capabilities in the Rational SSE platform enable reuse across engineering artifacts from the various parts of the lifecycle, as well as modular reuse of components across the entire system. The challenge to do efficient reuse is met by reusing components, applying proper change management across different variants, and leveraging automation machinery to control reuse and establish a robust variant control process.

Three architectural levels provide options that facilitate reuse scalability depending on the complexity of the system: multi-stream variant management, parameterization, and feature modeling.

The solution approach outlined in this article is open and provides a modular set of capabilities based on open protocols, so that different specialty systems engineering tools can integrate into the framework. The capabilities outlined here facilitate strategic reuse by lowering maintenance costs, improving time to market, and saving your company money.

**Acknowledgments and disclaimer**

We would like to thank Daniel Moul, and Ronnie Seagren for reviewing this article and providing many helpful comments. They helped to update it with recent changes in direction and terminology and contributed greatly to clarifying the content to its current state. We would also like to thank Steve Di Camillo, Bruce Douglass, Mats Gothe for reviewing early drafts of this article. Many thanks are due to Ran Rinat for the thought that went into the preliminary versions of this article.

The technical vision presented in this article includes intended future product direction and strategy.

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.

IBM Rational tools are designed to be part of your comprehensive software development process. As with any development best practice, you are responsible for developing and sufficiently testing your applications and related systems, including those related to safety critical functionality.

© Copyright IBM Corporation 2014
IBM Corporation Software Group, Route 100, Somers, NY, 10589, U.S.A.

Produced in the United States of America

All rights reserved

© Copyright IBM Corporation 2014
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)