**IBM Rational software development solutions**
**White paper**
June 2009

Rational® software

# Enabling software reuse using successful component-based development practices.

*Jean-Louis Vignaud, IBM Software Group, Rational*

**Contents**

## Executive summary

Software development techniques have evolved over the past 40 years from machine code to high-level languages and tools for system modeling and configuration. New technologies and platforms such as Java™, .Net, CORBA® and XML have helped practices such as Service Oriented Architecture (SOA), software reuse and Component-Based Development (CBD) become commonly accepted and practiced in the software development industry.

CBD offers a reuse-focused approach to the design, development, implementation and evolution of software applications. Software applications are assembled from components from a variety of sources with the components being written in several different programming languages and running on several different platforms.

These practices allow organizations to develop and manage very large, complex software projects, applications, and products which can be composed of several hundred components and hundreds of thousand source code files, developed or maintained by several hundred to several thousand engineers on multiple sites.

This paper describes four best practices examples where organizations have created and successfully deployed to handle this scale and complexity challenge.

- *Example 1 describes how a very large application composed of many components may be developed iteratively with a rapid development cycle in which all components follow the same release cycle.*
- *Example 2 describes how to handle the development of a large application integrating components that have their own release cycle as well as dependencies with other evolving components.*
- *Example 3 addresses the scenario where a very large project is divided into sub-projects that can impact any component of the software application and demonstrates management of parallel development streams and their continuous integration.*
- *Example 4 was inspired by open source practice and illustrates how components can be shared with multiple unrelated projects consuming and/or contributing to them.*

**Highlights**

***Component-Based Development (CBD) can help companies manage even the largest, most complex software projects.***

***CBD focuses on the effective reuse of design and development components.***

These best practices are supported by IBM® Rational® Synergy; The implementations with Synergy will not be described in this paper. For specific implementation information, see "Synergy process tailoring" training that describes how Synergy can be tailored to provide standard configuration management (CM) process patterns.

### Component hierarchy and single development stream

A product or system is composed of a component hierarchy. Each component can be developed and maintained by a different team.

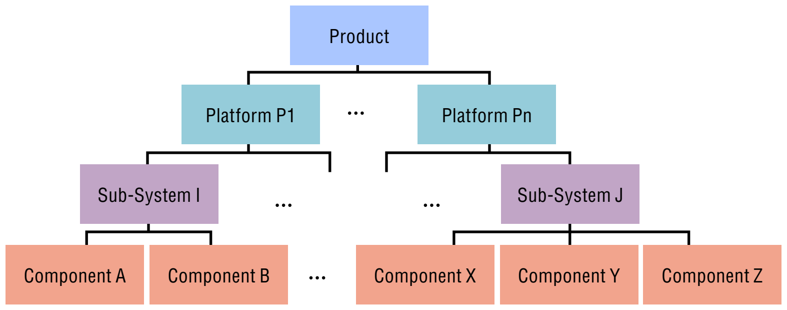

*Figure 1. A component hierarchy can be composed of a large number of component streams.*

***Use component hierarchy single development stream best practices to manage components on the same release cycle.***

The component hierarchy evolves in a single development stream, with a very rapid release cycle (one release every two weeks). Agile requirements management and iterative development ensure the component teams deliver these regular iterations.

### When to use this best practice

- *Specific teams in the organization have the component expertise ("expert" teams own key business components).*
- *Requirements for a component can be managed in a single development stream.*
- *Evolutions of the component hierarchy can be synchronized in a single common development stream.*

**Highlights**

This approach can be used as the development process of a small hierarchy (possibly a single component) or scale to large hierarchy of components; it is only limited by the number of changes that can be managed by the Release Control Board in each iteration.

*Successful implementation of this practice*
An IBM customer in the automotive industry implemented a single development stream process to control their component hierarchy and successfully managed the development and delivery of more than 70 components from more than 1000 developers dispersed over many development sites.

***Success story: An automotive customer improves productivity and reduces risk using a single development stream process.***

Thanks to the complete automation of this process with IBM Rational Synergy, this customer measured the following benefits:

- *Achieved process reliability and predictability.*
- *Increased their productivity.*
- *Drastically reduced the risk of making errors.*
- *Optimized their resource usage.*
- *Improved the company's efficiency by 33 percent overall.*

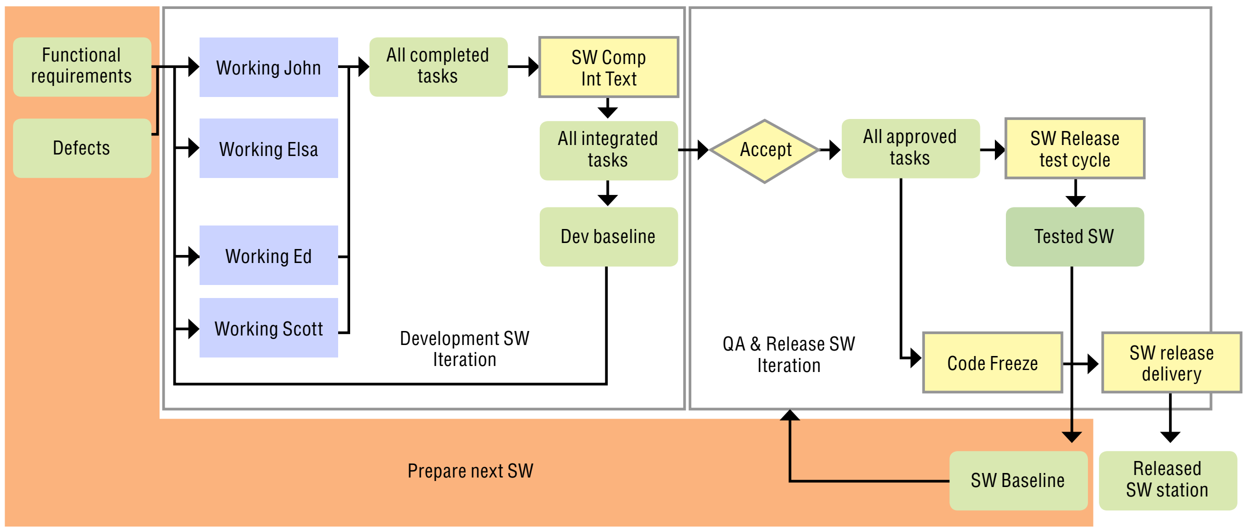

*Figure 2. Iterative development lifecycle with a bi-weekly release process*

**Highlights**

***Continuous testing and integration enable rapid development and greater agility.***

1. *Manage the agile requirements*

- *Functional requirements for an iteration are identified using the agile requirements management process and requirement prioritization.*
- *The definition of the requirements for the next iteration is done in parallel to the development activities for the previous one.*

2. *Develop components*

- *Development tasks are assigned to the development team at the beginning of the new iteration development phase.*
- *The development team is organized in sub-teams working on different software components for greater scalability and efficiency.*
- *The project manager monitors the development team members' progress toward their objectives and makes the necessary adjustments to ensure that the development goals will be achieved.*

3. *Perform continuous component hierarchy integration and testing*

- *The software is continuously rebuilt and tested to speed up the development process while keeping a consistent and stable software configuration.*
- *Developers work on a stable software configuration that they can update on a frequent basis to get the latest integration stage.*
- *The rapid integration cycle limits the number of parallel and concurrent versions.*

4. *Freeze a baseline*

- *At the end of the iteration (for instance, after 10 days of development), the release content is decided by the Release Control Board, and only approved tasks go through the software iteration qualification and release process.*
- *A software baseline is created and the development of the next iteration starts immediately.*

5. *Perform quality assurance tasks*

- *The 2 to 4 day QA and release cycles are performed in parallel with the development activities for the next iteration.*
- *The quality of the iteration is validated through appropriate functional and operational testing.*
- *Developers make the necessary hot fixes which are added to the build and validated.*
- *When it passes QA, the iteration is released.*

6. *Release the component hierarchy*

- *An iteration is released every 10 days.*
- *It usually takes 12 to 14 days from deciding of requirements implementation, or deciding of defect fixes to release.*

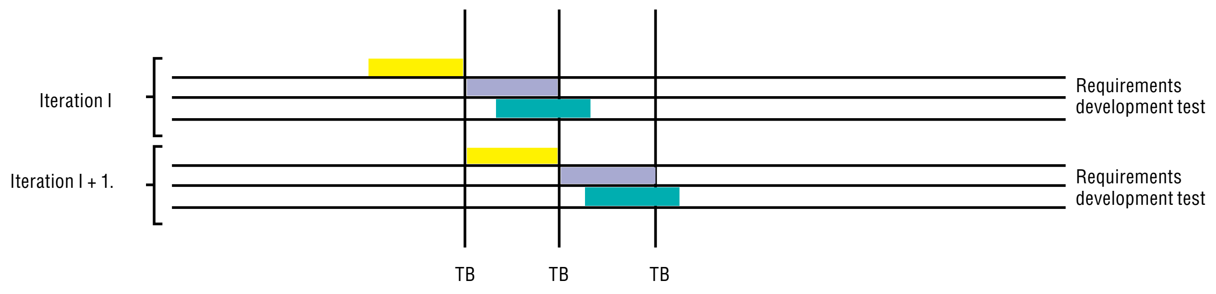

*Figure 3. Parallel development of each iteration and no downtime for the requirements engineers, developers and testers.*

**Highlights**

***Using component hierarchy best practices, there are no process interruptions between development iterations.***

***Use component hierarchy multi-layer development streams best practices to manage components on different release cycles.***

This development process avoids development team downtime because:

- *Developers are always working towards the "next" iteration:*
  - *They do not have to worry about in which SW iterations their tasks will to be included.*
  - *They do not have to worry about creating new development workspaces for the next development iteration.*
- *There are no process interruptions of development activities between 2 development iterations.*
- *Every 2 weeks during the 2 to 4 days of the iteration release process, developers may be asked to perform a hot fix for which they have a readily available design environment.*

**Component hierarchy and multilayer development streams**

In more complex projects environments, components may have a different release cycle than their consumers. This is often the case when components have many consumers and complex dependency relationships. In such cases it is not possible to coordinate the evolution of the product and its entire component set in a single development stream.

It can be difficult to complete a build in these complex environments because the dependencies between the components are often broken. Therefore, it can be difficult for a build manager to produce a consistent build with little guidance on the complete set of components needed to bring the entire stack together.

**Highlights**

***The multi-layer component stream approach is highly scalable for large, complex projects.***

***Success story: medical device company significantly improves their application build success ratio using multi-layer component streams.***

If your organization is faced with these challenges, it's important to secure a higher level of control in order to be able to promote independently managed components up the stack while maintaining build stability. This control can be obtained using multi-layer component streams:

- *A development stream can correspond to a single component, or correspond to a group of components that evolve together.*
- *Each component is modified in a single development stream.*
- *A hierarchy of development streams defines the workflow of changes through bottom-up promotion path: Changes performed in lower level components are promoted to their higher level consumers.*

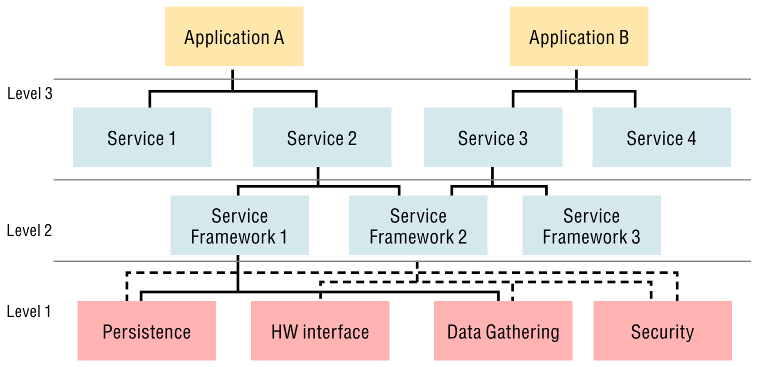

*Figure 4. "Persistence" component is consumed by two higher level components; changes on "Persistence" component are promoted to all consumer components belonging to the higher level at the same time.*

*When to use this best practice*

- *The components developed have many consumers.*
- *Components have independent and possibly uncoordinated release development cycles.*
- *Requirements for each component can be centralized and prioritized.*
- *Each component has a dedicated team with complete responsibility for its development.*

The multi-layer component stream approach is also highly scalable. If the team grows significantly, it may be difficult for other layers to get the correct component levels to consume. Organizations may add an integration build management team to ensure that the promotion up the layers is done effectively. Creating such a team would transfer Consumer Integration Baseline responsibilities to a coordinated group across components.

*Successful implementation of this practice*
An IBM customer in the pharmaceutical and medical device industry maintains a multi-site environment (four sites) that includes large-scale component development with multiple layers (four or more). The propagation of changes up the component stack needed to be carefully managed. By implementing multi-layer component streams, the customer achieved the following benefits:

- *Significantly improved the application build success ratio. Build failures due to consumed components inconsistencies are now exceptions that may be resolved quickly.*
- *Obtained predictable control over how changes are promoted across a very large component framework application. Specifically, all components work exactly the same across distributed sites.*
- *Achieved high scalability by being able to roll up changes from lower level components with minimal effort while still retaining management control.*
- *Improved project oversight by enabling managers to identify, prioritize and communicate task and timeline information to teams.*

**Highlights**

*Process description*

- *Each component has its own development, integration and testing cycle.*
- *Each component has a build manager responsible for producing quality iterations of the component. These iterations are available for use by the component consumers.*
- *Deliveries of component iterations on the same layer may be synchronized to deliver a consistent set of iterations that can be used together by any component consumers up the stack in the application.*
- *Deliveries of component iterations include both changes performed for the component itself as well all changes made on the consumed components, thereby ensuring consistency of the delivery.*
- *Consumers always integrate deliveries from lower level components with their latest changes. Code is not "merged upstream" until it has been thoroughly tested, to minimize broken builds.*

***Use the continuous development streams integration best practice if your project is comprised of multiple sub-projects being developed in parallel.***

**Continuous development streams integration**

The above processes define instances where a component is modified one development project at a time. Therefore, there are no parallel evolutions of the components that need to be merged. These processes work well when it is possible to centralize the requirements for the next evolution of the component. Such approaches may not be possible on very large-scale projects.

Very large-scale projects may be broken into:

- *A component hierarchy as described in the examples above,*
- *A collection of smaller, parallel sub-projects on the same software.*

**Highlights**

***Parallel development streams enable teams to ensure applications created independently can be eventually integrated together.***

Each sub-project has:

- *A functional objective.*
- *A development team.*
- *A project manager completely responsible for the project delivery.*
- *Its own development stream with the need to follow an Agile process so it can be delivered quickly.*

In this case, many different and parallel development sub-projects can impact any component of the software application or product, and there is a need to manage parallel development streams.

*When to use this best practice*
Use parallel development streams when:

- *Many ongoing functional changes need to be performed in parallel and a component can be modified by several projects at the same time.*
- *Each project wants a full control on everything on the path to releasing their product.*
- *The projects have to be eventually integrated altogether in order to build the delivered software.*

*Process description*
In this scenario, no established hierarchy of projects exists. All projects must be integrated for the formal application or product delivery.

Each sub-project is developed independently, and a process is created to ensure that changes delivered by the other sub-projects are continuously integrated. This method ensures that parallel development performed in different projects are limited in number and merged regularly. Continuous integration enables an iterative development process for the product.

The following is an overview of the iterative integration process:

- *At the beginning of each development iteration, a single sub-project delivers its iteration to the parent project. For each iteration there is a different sub-project doing the delivery, which is governed by a formal schedule identifying when each sub-project has to make a delivery to the parent project.*
- *The parent project verifies the delivery quality for the whole application/product (the sub-project may only work on the sub-set) and publishes accepted products it to the other sub-projects.*
- *Sub-projects immediately perform the integration of the delivery published by the parent project.*
- *The process mandates that any sub-project delivery to the parent project includes all previous deliveries from the parent project. This way the integration process limits the scope of the merge:*

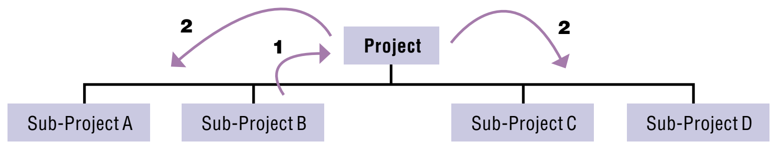

Figure 5. Sub-project B delivers its iteration to the parent project (step 1). When the delivery is approved it is published to the other sub-projects (step 2) that will have to integrate it before they can deliver to the parent project.

- *Each sub-project receiving the delivery of the parent project has to merge it with its own changes.*
- *The sub-project that has delivered to the parent project and initiated the new parent project delivery has no code merge to perform in this iteration.*
- *Same merges are not duplicated among sub-projects.*

The master integration for the sub-projects development process is as follows:

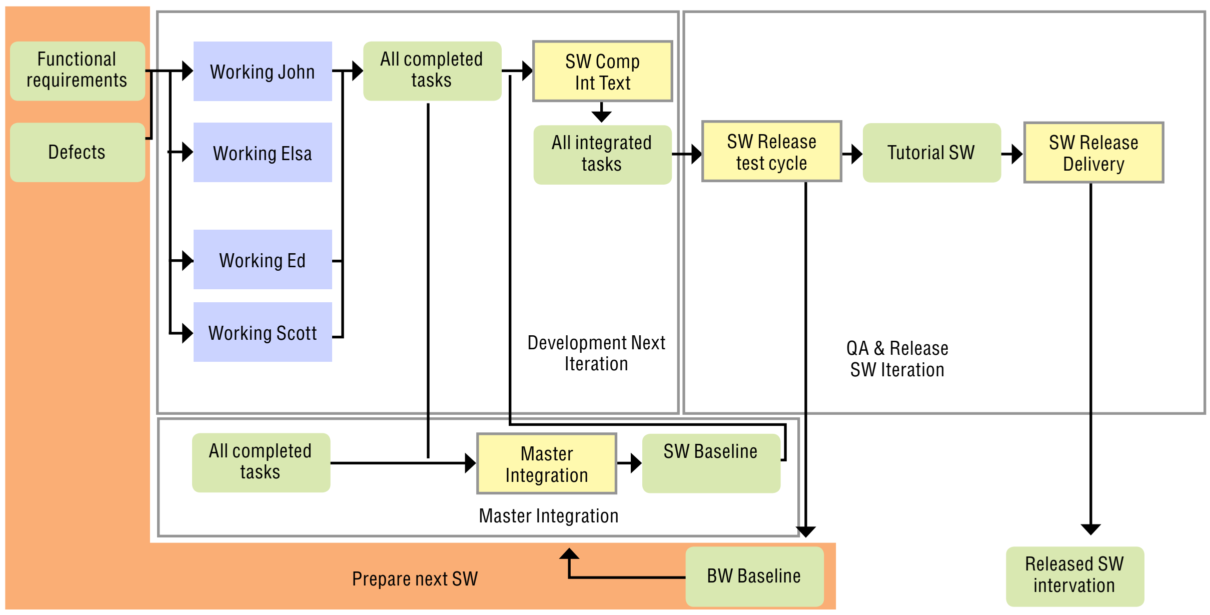

*Figure 6. Development process of a sub-project*

The objective of such a process is to integrate deliveries of sub-projects, and to alternate which sub-project delivery is integrated. It gives you flexibility on the following:

***A master integration process gives you greater flexibility in determining your release cycles and planning your integration timeline.***

- *The length of a release development cycle for each sub-project – teams only have the requirement to finish a release development cycle when they have to deliver to the parent project.*
- *Selecting when you are ready to integrate the parent project delivery (For example, when the deliveries of the other sub-projects that have been accepted by the parent project).*

It limits the merge activities to the sub-project development streams, and no merges are required in the parent project. It requires sub-projects to workout and agree on a delivery schedule plan for the parent project that will have to be strictly respected by each sub-project so the process can handle an integration cycle on a frequent basis (for example, every two weeks) and deliver the expected benefits of frequent integration (fewer merges).

**Highlights**

To optimize the process and to further limit the number of required merges, the sub-projects can agree on when shared components are updated in each sub-project. This way they could limit the parallel development on a shared component by ensuring they have received the changes made on a component by other sub-projects (through the continuous integration process) before modifying it.

***Shared components are a good fit when developing families of products that need to change and grow together***

## Shared component repository

In the previous models, each component is generally owned by a specific team, which controls its evolutions and implements the requirements of the component's consumer.

This approach lacks flexibility when organizations developing families of products need to be more reactive to customer requests. Projects may have prioritization conflicts for their requirements on a component. To address this, a more collaborative environment is needed, allowing each project to modify the components they need in a more flexible way.

The shared component repository approach is inspired by Open Source practices. Shared components are in a central repository and may be modified by any component consumer that requires it.

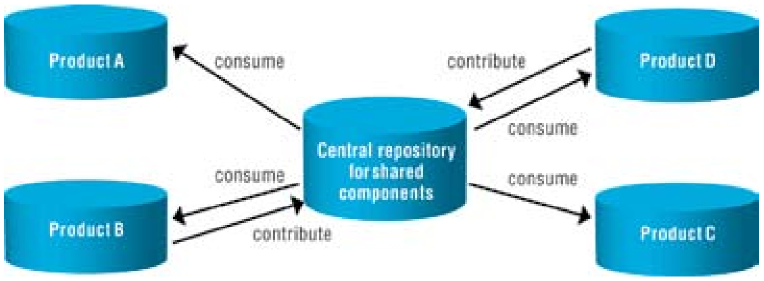

*Figure 7. Central repository for shared components and delivery of shared components to consumers and receive contribution on shared components*

An architecture board defines and manages a roadmap on the shared components, and assigns (temporarily) a component to a project team. A shared component may be modified in one project at a time.

*When to use this best practice*

A shared component repository process is a good fit when:

- *The organization is developing product families that share a common set of components.*
- *There are independent projects that reuse and possibly require modifications on shared components.*
- *It is not possible for a central team maintaining the shared components to cope with the change and adaptation requests.*
- *Project managers want full control of everything on the path to releasing their product.*
- *There is a strong focus on project execution – shared components must not get in the way of delivering the projects.*
- *The organization is open to new development techniques such as those established in the Open Source community and accepts coordination of shared components changes.*
- *Development teams have the appropriate skills to update shared components or can get temporary access to resources with the appropriate skills.*

*Successful implementation of this practice*

An IBM customer in the consumer electronic market implemented a shared component repository approach based on lessons learned from the Open Source community to further improve their reactivity to market requests by removing collaboration conflicts on shared components.

**Highlights**

As a result, the organization measured the following benefits:

- *Increased quality and productivity due to the implementation of a central repository for shared components.*
- *Better focus on business objectives for all teams.*
- *Faster turnaround time for customer requests.*
- *Increased collaboration and contribution to the evolutions of shared components.*

***Success story: a consumer electronics company improved their turnaround time on customer requests and increased their collaborative capabilities by implementing a shared component repository development approach.***

*Process description*
The shared component repository approach uses principles that mirror those found in large Open Source projects:

- *No dedicated team for the shared components.*
- *No (permanent) component ownership.*
- *Component customers who find bugs, need new features, new components become contributors to the consumed components.*
- *When changes on a component are performed and verified, they are published on the repository, so they become available for the other projects.*

An architecture board composed of the architects from the various customers and contributors defines and updates the roadmap for the shared component:

- *What is / What should / What will be in the repository?*
- *What is required on components? Who uses them?*

An executive board decides which project can modify a component and helps avoid unnecessary parallel modifications:

- *Explicit separation between component interfaces and implementation.*
- *Permission to fix the implementation does not give the right to modify the interface.*

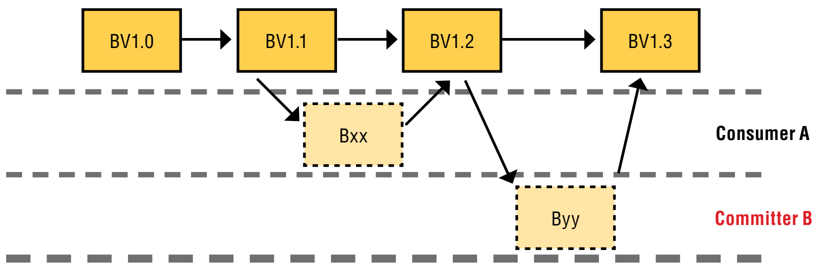

*Figure 8. A component has only one committer at a time, the committer (e.g., project modifying the component) is changing over time based on the executive board decision.*

*Process variant*

In this variant there is no need for an architecture board and executive board, enabling the teams consuming the components to tailor them as they need, when they need. This is achieved by having a team responsible for each component.

Open source terminology introduces three types of stakeholders:

- *Committer: the one that updates the component. In that case the committer is the team responsible for the component.*
- *Contributor: the one that makes modifications suitable for the general need and that proposes them for inclusion in the next version of the component.*
- *Consumer: the one that uses the component and possibly updates it.*

**Highlights**

***A variation on the shared component repository process enables teams to tailor components as they see fit, without board interaction.***

The process is therefore:

- *A component is owned by a team (the committer). The team owning the component is not in charge of making all the modifications in the component source code. Instead, the team's responsibility is to release new versions of the component on a regular basis with appropriate quality levels.*
- *Teams consuming the components can modify (tailor/bug fix) the components as needed. Sometimes, consumers have a better understanding of tailoring requirements than the team owning the component and can more productively update it.*
- *The teams consuming the components and modifying them are invited to become contributors. This means they are invited to deliver their modifications to the team responsible for the component in order to have these modifications included in the next version of the component. To do this, they must follow strict guidelines when updating the consumed components.*
- *Each consuming team has to merge their modifications in the new component versions when delivered by the committer. Avoiding this merge effort is a strong motivation for becoming a contributor.*

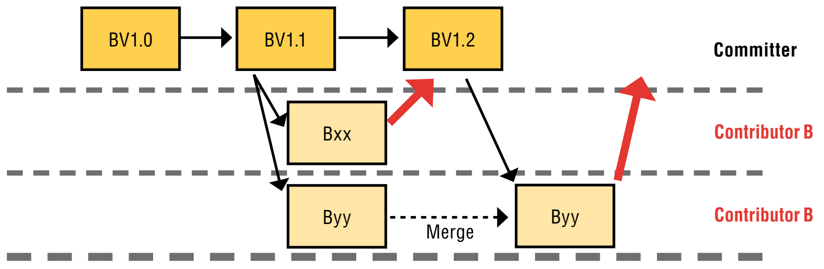

*Figure 9. Projects modifying the component are motivated to become a contributor in delivering their changes to the committer for inclusion in the next component version in order to avoid merging their changes in the new component versions when delivered.*

This process may yield the following benefits:

- *Create increased agility for the project teams who are free to modify reused components as needed.*
- *Enable teams owning components to release more often and with more content as they receive changes from the contributors.*
- *Increase overall satisfaction with the reused components and enable a significant increase in organizational productivity and quality.*

## Conclusion

Although software projects are growing exponentially in size and complexity, implementing CBD best practices can help you gain control of your development initiatives and maintain agility while meeting time-to-market requirements. Implementing a structured project architecture with an appropriate workflow can enable autonomous but coordinated teams to design even the most complex software systems without sacrificing efficiency, innovation or agility.

By implementing the best practices enabled by advanced change and configuration management solutions such as IBM Rational Synergy, you can more easily design, manage and reuse the specific process patterns supporting the complex needs of your organization today and help you compete and thrive into the future.

## For more information

To learn more about advanced change and configuration management best practices and IBM Rational Synergy, please contact your IBM representative or IBM Business Partner, or visit:

**ibm.com**/software/rational

RAW14135-USEN-00