OpenText™ ALM

Entities Sharing Best Practices

For ALM Practitioners

opentext™

Legal Notices

The only warranties for products and services of Open Text and its affiliates and licensors (“Open Text”) are as may be set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Open Text shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Disclaimer

Certain versions of software and/or documents (“Material”) accessible here may contain branding from Hewlett-Packard Company (now HP Inc.) and Hewlett Packard Enterprise Company. As of September 1, 2017, the Material is now offered by Micro Focus, a separately owned and operated company. Any reference to the HP and Hewlett Packard Enterprise/HPE marks is historical in nature, and the HP and Hewlett Packard Enterprise/HPE marks are the property of their respective owners.

Contents

Welcome To This Guide

Welcome to the Entities Sharing Best Practices guide.

This guide provides concepts, guidelines, and practical examples for the best implementation of entities sharing such as requirements, tests, test assets, and business components in various organizations.

About Entities Sharing

Traditionally, the application quality management market has been focused on specific testing activities, like load/stress and functional/regression. As the market shifts, organizations are seeking a combination of greater business value and agility. Businesses are therefore required to take a more holistic and agile approach if they hope to attain better quality. To meet their business objectives, they need to evolve from finding defects to validating functionality. Such an approach is driven by the need for companies as a whole (not just their IT divisions), to increase efficiency and productivity so that they can maintain market leadership and attain a competitive advantage.

The organizations of today must validate and manage:

- Business functions

  Raise user acceptance level and decrease testing costs while providing maximum requirements coverage.

- Production readiness

  How does the application scale? Is it secure?

- Risk status

  Increase regulation (for example, data privacy, and compliance) and security requirements while managing costs.

To manage the process, organizations need to focus on overall governance rather than specific tasks. In addition, an increase in the complexity of modern systems, composite applications, and Service-Oriented Architecture (SOA), as well as the growing number of regulations requires more effective solutions to satisfy quality levels.

Application Lifecycle Management addresses these needs in a comprehensive manner. It facilitates all types of application tasks, using a solid foundation for managing complex initiatives and regulatory requirements. ALM is designed to address the needs of organizations that are looking to track and manage projects of all sizes, including large initiatives and enterprise-wide releases. ALM can help facilitate a Center of Excellence approach - a logical or physical entity that drives standardization and processes across an organization, to improve quality, consistency, effectiveness, and efficiency.

The purpose of this document is to assist ALM customers to assess their current SDLC practices and successfully build and maintain a quality methodology using advanced features provided by ALM. All aspects of this process have been researched using best practice data and expertise from various sources including OpenText's operating system administrators, professional services organization, technical documentation, books from industry experts and personal experience of many customer quality organizations. These

guidelines will help reduce in initial creation time and achieve maximum value in operating the ALM.

Audience

This guide is intended for:

- Business Analysts
- Quality CoE Managers
- Quality Automation Engineers
- Development Managers

Prerequisites

To use this book, you should have a good acquaintance with major phases of Software Development Life Cycle (SDLC). You should also be familiar with the business processes in actual IT organizations.

Operational knowledge and administrative privileges of ALM are essential in implementing these best practices. It is strongly recommended to read *Libraries and Baselines*and *Imported Libraries*Chapters of *Application Lifecycle Management User Guide* to get general understanding of libraries, baselines, import and synchronization features mentioned in this document.

Note: In Quality Center Starter Edition and Quality Center Enterprise Edition these features are disabled.

Structure

This guide is organized as follows:

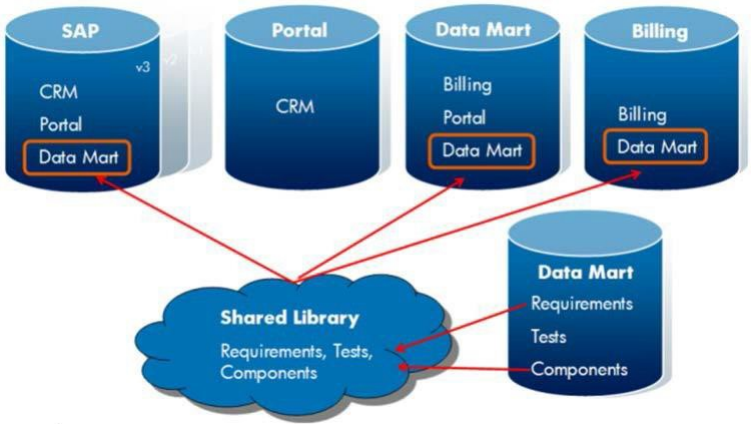
- *Introduction To Entities Sharing*
- *Preparing for Entities Sharing*
- *Making Entities Sharing Work*
- *Conclusions*

1 Introduction to Entities Sharing

Importance of Entities Sharing

In the world of cross-functional IT initiatives, the same application may be incorporated into multiple initiatives. It is critical for various IT teams to be able to share the same set of requirements, tests, test assets, and business components in order to validate that the new initiative is working as expected and does not disrupt the behavior of other applications. Instead of recreating entities, customers should be able to share the same entities across projects.

Since many industries are heavily regulated and must pass a variety of compliance-based tests such as SOX and HIPAA, it would be highly inefficient to duplicate entities across multiple projects especially when these regulations are updated regularly.



Another factor affecting the behavior of IT organizations is agile development, the centralization and global distribution of project teams. In general, conditions for building products have evolved from performing some isolated steps of Software Development Lifecycle (SDLC) to integrating all steps of the product's life cycle – regardless of the approach taken. Business analysts are now more involved at all stages of the development lifecycle. They prepare general requirements and continuously validate the requirement coverage and product readiness for production. The project management staff requires better access to the status of the entire process during each stage of the project. Developers need to understand how their bugs impact the project schedule.

The reusable library of requirements, tests, test assets, and business components allows multiple stakeholders to coordinate their efforts to support enterprise-wide releases. Each team can import the baseline of the library and work with the assets within their own project.

There are two types of libraries:

- A *source* library is a library used as the basis for creating another library.
- A *destination* library can be created by importing an existing source library from the same project or from a different project.

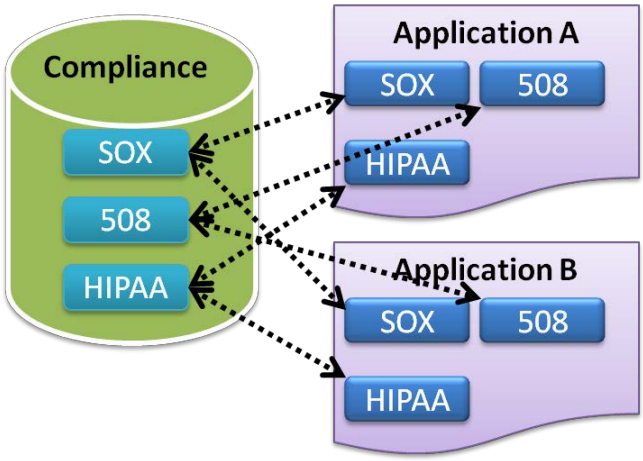
Changes made to entities in an imported library, either in the source project, destination project, or both, can be viewed from either project. If desired, those changes can be synchronized between the two projects to enable ongoing consistency. A library also enables the organization to reuse compliance requirements and tests, reduce duplication of effort,

and aggregate metrics across projects using a common set of metrics.
Alternatively, even if all assets are maintained inside one large project, using entities sharing allows independent work of multiple stakeholders while ensuring consistency of the data.

Common Entities Sharing Scenarios

Different companies manage various areas of SDLC (such as requirements management, testing and defects tracking) in a similar manner. Most companies go through the same path of maturity, while some reach the higher level and others stop short in the comfort zone of lower effort. No matter what the level of maturity, when it comes to development processes and scenarios, companies usually go through these processes:

- **Compliance**



Many organizations—especially those in the finance, healthcare, and government sectors— are required to comply with specific government regulations such as HIPAA, Sarbanes- Oxley, US Section 508, and many others. It is therefore obligatory for their IT divisions to demonstrate adherence to the highest level of regulatory compliance. The following steps are usually required by these regulations:

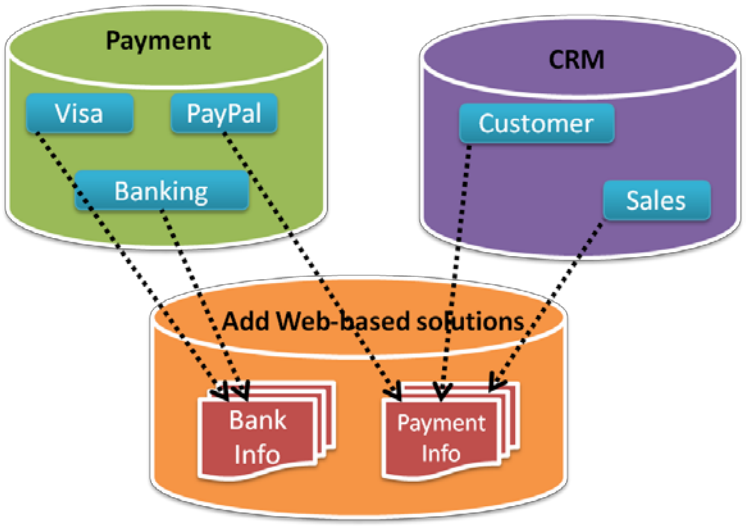
— Sign-off of certain processes and documents after passing necessary reviews and approvals. In a regulated environment, businesses must provide proof and reasoning when they make decisions that may potentially affect compliance with the law and standards.

— Generation of reports in predefined formats that provide sufficient proof that the organization meets the required level of compliance with specific government or industry regulations and requirements.

— Auditing of changes that impact regulatory requirements throughout the application lifecycle to show application consistency.

— Propagation of changes to all repositories that use a certain application.

- **IT Initiatives**



Mature IT organizations often adopt an approach called business initiative. An initiative is a set of applications that are developed to support a common business goal. An example of an initiative is adding web-based payment solutions such as PayPal to an existing credit card clearance process. To support this initiative, several new applications may need to be developed and a large number of the existing software applications may need to be modified.

The scale of these developments and modifications could not be managed if each application team works alone in their respective silos. Instead, all requirements management, development, testing, deployment and change management processes need to be coordinated across multiple applications. With each application team introducing their own requirements, tests, components and defects, the key to a successful release of an IT business initiative is to enable visibility, coordination, and collaboration.

There are some major challenges in the implementation of an IT initiative that are addressed by ALM:

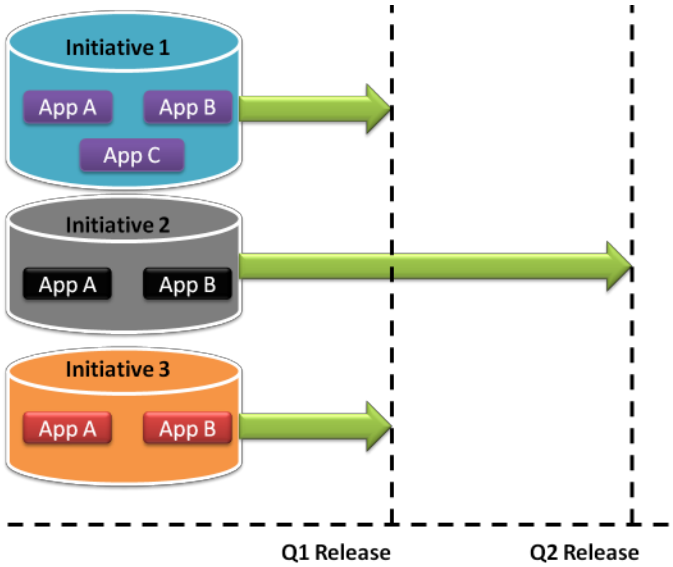
— Progress reports provide the vital understanding of the IT initiative's current status at any point in time. Multiple applications and even other related initiatives may depend on the requirements or testing progress of a project. If each application participating in the initiative is represented by a library, then ALM provides an easy way to import several libraries into a single project to manage this initiative.

— Proliferation of composite applications dictates the need to test not only each component individually but also end-to-end business transactions and communication between all components.

— With so many composite applications and increased SOA usage, one small change in the requirements or a defect fix may affect the entire business transaction. Change impact analysis of the initiative and its hidden dependents across all applications is imperative in reaching "go"/"no go" decisions early in the cycle.

— If the same application exists in multiple repositories, all relevant stakeholders must make sure that any changes to an application are properly synchronized between multiple instances of the application.

- **Enterprise Release**



A common approach among mature IT organizations is to group multiple initiatives together and release updates periodically – monthly, quarterly or bi-annually. Each release consists of a number of IT initiatives—each with its own requirements, dependencies, timelines, and priorities.
Managing an enterprise release is complex: it involves tracking and reporting the progress of each of the initiatives and their underlying applications. By sharing entities, the enterprise release management team can control the data consistency and quality of each of the "ingredients" of the final package. The release of multiple initiatives requires even more visibility, coordination, and collaboration. Multiple stakeholders must be able to assess the quality of each of the inter-connected projects and measure the overall release status and readiness at any time.

- **Parallel Development**

To respond to the demands of a hyper-competitive marketplace, IT departments today are tasked with increasingly diverse responsibilities. They must support global, 24x7 operations and integrated supply chains while quickly delivering applications to market.

To support this, companies often build software by employing parallel development in one of two modes: the waterfall approach or agile development. In the waterfall approach, each team may develop separate, unrelated features and deliver them at the end of the process for final integration.

In agile development:

— development teams must manage an increasing number of rapid changes,

— business analysts need to learn how to capture requirements in higher- level user stories which are both more flexible and can be easily interpreted by test and development teams,

— QA organization needs to ensure they are prepared to test the multi-layered applications delivered by development. Their test plans need to be flexible enough to accommodate the changes in each iteration.

ALM provides tools to achieve these goals including entity sharing between projects being developed *in parallel*, regardless of what approach is taken – waterfall or agile - thus allowing quick synchronization of changes and traceability.

2 Preparing for Entities Sharing

This chapter describes the basic steps for best using ALM for entities sharing.

Definition of a Project

The first step in implementing ALM is to decide what the ALM project should represent. This decision will have a profound effect on the way the project is managed, how requirements are written and the approach that is taken to test the application. Since no two IT shops are identical, the decision depends largely on considerations such as the company culture, business processes in place and previous tool limitations (if any). Most ALM projects (database with resource repository) can present one of the following:

- **Application**

  This is the common case. Each project represents one application, with all of its requirements, business models, KPIs, test sets, test resources, defects and reports. This is the natural way to map development activities, to manage relationships between business analysts, developers and testers. For example, Billing, CRM and Portal are widespread names of projects and these are dedicated to their namesake applications. The situation may get more complex when one application is used as a component of another application or initiative.

- **Version/Model**

  A major application version generates a new ALM project and serves as a starting point for future development. This is common in ISV circles where a major version signifies the end of a big cycle and start of a new one. In this case, the older version enters a maintenance state, and the new version moves through development milestones towards a release.

  ALM is implemented in various types of companies. Sometimes a new project represents a new model (physical entity) of the product, such as a cell phone, printer or TV set. In this case too, a new project is created when new model development is started.

  Consider the fact that the same goal may be achieved by using the *Application* usage scenario and employing release and cycles features of ALM to manage new versions.

- **(Custom) Project**

  This is also a common usage pattern found in many IT organizations. For example, suppose there is a requirement to create a web application to display the credit rating of a banking customer. To develop this new feature, a new project is opened to cover all aspects of planning, coding, testing and bug tracking. Integration of existing components and/or systems usually falls into this category.

- **LOB**

  In a world of huge companies and endless mergers and acquisitions, one Line of Business (LOB) may be quite separate from another or just maintain independent sets of data, thus making the case for this kind of implementation. Each LOB maintains one project where it keeps all applications or compliance information.

- **Repository**

More frequently than not, stakeholders in the application lifecycle elect to designate one project as a repository/rollout so that master records of requirements, test, test sets are kept there. While not a "project" in strict terms, it gives the ability to maintain the "master definitions" part of the process while all "execution" aspects are stored in other projects.

We strongly **recommend** having one repository project when developing and testing complex or composite applications.

- **Everything**

Sometimes the company decides to maintain all activities of all teams inside one rather large project. This type of implementation may indicate the desire to keep everything in one place and separate various types of entities by granting certain privileges to certain users. Alternatively, such projects may be dedicated to one of the various ALM activity types, for example, a defects-only project or requirements-only project for all kinds of applications.

We would **not recommend** implementing this scenario due to the contradiction with the ALM project management capabilities (by default, the project should have a clear start and end) and possible complexity which would increase over time.

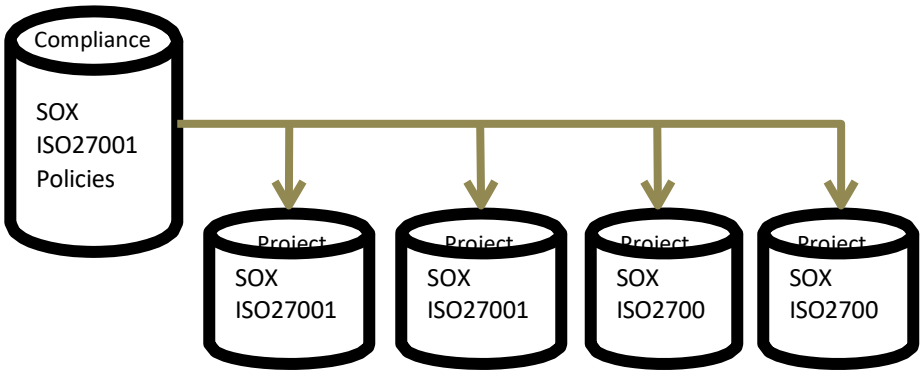
ALM allows great flexibility in mapping software development processes – therefore it is imperative to analyze them prior to implementing the tool.

Library Types

A *library* represents a set of entities in a project and the relationships between them, such as *coverage* and *requirements traceability*. The entities in a library can include requirements, tests, test assets, and business components. After creating a library, you also create a *baseline* in order to have a snapshot of the library at a specific point in time. You can compare baselines at all stages of the application lifecycle. Viewing baseline history enables you to track changes made to individual entities in your library over time. As development continues, you can view and compare all versions of an entity that are stored in a baseline.
Based on comparison, you may decide to import a baseline to reuse an existing set of entities, or you may choose to synchronize the projects to maintain the same level of functionality.

The following sections describe different library types based on the usage *scenarios* and *origin* of the project:

- **Compliance Library Type**



This library type primarily aims to provide reusable compliance requirements and generic test case definitions through sharing in companies that need to address regulatory rules that change over time such as Sarbanes-Oxley, HIPAA, COBIT, and many others. The library is usually defined as read-only after its initial creation by importing from the repository/rollout project. The Compliance Library Type is locked down via permissions to allow for the extraction of the requirements to the project library but not modification of the compliancy library itself or the extracted requirements. When a compliance

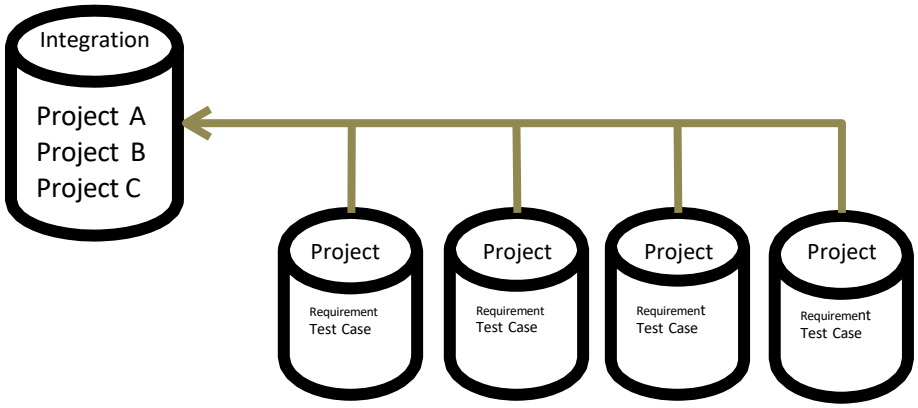
requirement needs to be updated within the repository itself, the Compliance Officer (or someone fulfilling that role such as Library Administrator) makes the modifications within the library. The changes are applied to the implementation project using synchronization - a changed icon and alerts let the project owners know they should resynchronize the requirements. A full audit history of library changes is provided by ALM.

We **recommend** creating a new baseline, ideally with the date of change in the title, upon modification of the library requirements set. In this case, change request and exception request workflow are usually established. When a Compliance Library Type is in use, collisions are not possible.

We also **recommend** keeping a library per each specific compliance requirement to allow for maximum flexibility. This way you get only the entities relevant for the selected regulation when importing a library and/or synchronizing between libraries.

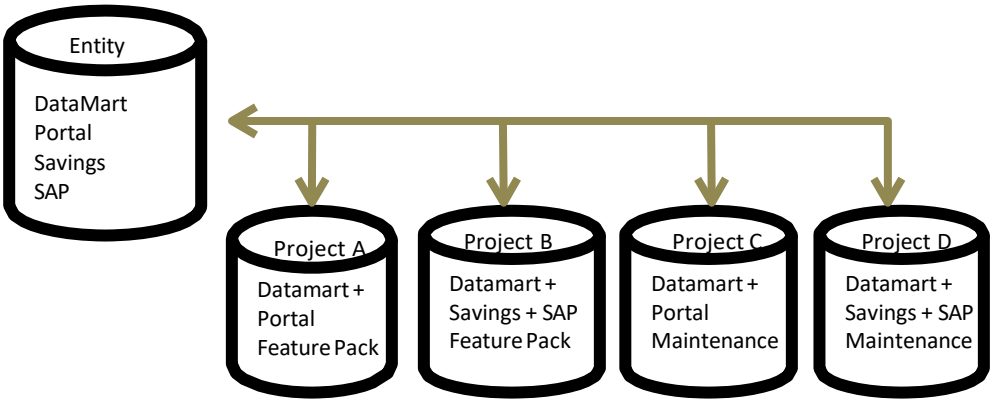
See a detailed explanation of development *scenarios* and participant *roles* later in the document.

- **Integration Library Type**



This library type's goal is to provide reusable requirements and test cases for one-time usage through sharing. It is often used as a repository for common requirements and artifacts that span multiple projects, for example, common GUI interface test set, common database or security requirements, performance requirements/test and so on. This type of library acts as a "template" library and as such, it is defined as read-only after its initial creation by importing from the repository/rollout project. Change request and exception request workflow are rarely established in this type of library. Changes applied in the integration project are seldom synchronized with repository/rollout project and vice versa. With this library type, collisions are possible but usually discarded in both the implementation and repository/rollout projects because they can corrupt the library.

- **Entities Library Type**



This library type is intended to manage reusable requirements and test case definitions that are considered assets to the application. As opposed to other library types, this library is typically defined as read\write after its initial creation by importing from the repository/rollout project. The workflow for change requests and exception requests is established according to the adopted methodology. Changes are made only to the implementation project, and the changes are applied to the repository/rollout project using synchronization. With this library type, collisions are possible and need to be handled to avoid the corruption of the library.

These are suggested library types – in actual implementations their usages can vary.

Defining a Library

Library structure should be planned with future needs and growth in mind. We **recommend** examining *granularity* of the libraries before creating them.

Smaller libraries give greater flexibility in assembling multiple combinations of assets for sharing. On the other hand, too many libraries may cause management overhead and confusion. When creating libraries, you may want to use *filters* to select only the relevant informational resources instead of selecting generic roots. This model gives the user more control over library content, and helps to define libraries that are not based solely on the hierarchical structure of the project. Another approach is to define "initial roots" for the libraries and let ALM automatically gather all of the relevant entities based on predefined links (for example, coverage and requirement traceability).

We **recommend** synchronizing baseline creation time with a major step in the development process such as the end of a cycle, iteration, or release.

When creating a library, you may prefer to choose requirements coverage and/or test coverage options that may result in a large population of linked entities even if you selected a small number of records. To avoid performance problems, the number of entities **recommended** for a single library is calculated based on two site configuration parameters:

- REQUIREMENTS_LIBRARY_FUSE with default value of *3500*
- Maximum number of requirements in a library should not exceed this parameter value, i.e. 3500 entities
- LIBRARY_FUSE with default value of *2500*
- Maximum number of tests in a library should not exceed this parameter value, i.e., 2500 entities
- There is a ratio of 1:4 between tests and resources, i.e., maximum number of resources should not exceed the quarter of LIBRARY_FUSE, i.e. 625 entities
- Same rule works for business components, i.e., the maximum number of business components should not exceed the quarter of LIBRARY_FUSE, i.e. 625 entities

These values are verified when you create baselines, import libraries, or synchronize libraries. We strongly **recommend** limiting the number of entities in a library to the sum of all sorts of records according to the rules above.

Importing and Synchronizing a Library

We **recommend** using cross-project customization to make sure that the fields used in all your projects have common definitions, making it easier to import and export projects.
Shared customization is the preferred way to introduce custom fields to projects, otherwise they are ignored by the import and synchronization process. Put all custom fields and values in a template project and use this template to create new projects. There are more reasons to establish a template project, such as cross-project reporting and defect synchronization.
As shown in the previous *section*, in the majority of implementations customers have more than one project to cover their needs. Having a template project will help in proactively reducing problems regardless of the approach taken.

The process of importing the library can be time-consuming. As the import operation builds the library structure and populates every entity, folder and link, large libraries could take several hours to import. Moreover, during the import process, the structure of the library is unstable as more entities are being created and populated, so we **recommend** not using the library objects until the import process is complete. An easy and **recommended** way to accomplish this is to perform imports of large libraries overnight, when ALM is not being used for its normal activities.

Similarly, when synchronizing the library that contains entities that are under version control, it is **advisable** to verify that all objects on the destination side are checked-in and not locked.

The first step of the library import or synchronization process is verification. The verification process includes the following checks:

- Requirement type check. Check that your project contains the necessary requirement types.
- Entities compatibility check. Checks that your project has the necessary extensions enabled. If the source project has an extension enabled, and the source library includes entities for that extension, your project must also have that extension enabled.
- Library size check. Checks that the number of entities in the library does not exceed the maximum defined by site configuration parameters (see *explanation*).

The synchronization process is performed based on these rules:

- If an entity was changed in both the source and the destination libraries, the source entity gets overwritten. The change is recorded into the history log of the entity. If the source entity was not changed, but the destination was modified, the destination entity remains untouched.
- When an entity is deleted in the source project, it is placed into the special *obsolete* folder in the destination project. This allows additional verification before final deletion, or it allows entities to be restored to their original place in the hierarchy. After reviewing the obsolete folder, the user should remove any entities—only then is the synchronization process complete.
    - The process does not allow schema inconsistency and will fail during the verification step even before any data is copied. It is therefore important to keep source and destination projects entities in the same shape in terms of schema. To ensure consistency, use cross-project customization as described above.
    - There is no way to merge data between source and destination entities if there is a data conflict. These conflicts may be a result of the baseline version of the entity taking precedence over the locally modified copy of the same entity. In this case, the only way to merge the changes is to do so manually.

Import Status

Import and synchronization processes may become quite complex, especially when there are many projects participating in the process. To better understand the status of the entities and the impact their change or deletion may have, open the "Usage" view, Imported By/From tabs. See *Imported Libraries* Chapter of *Application Lifecycle Management User Guide* for details.

Another option for checking the entities status is to generate a Baseline Report. This report provides detailed information on the baselines content in a standard ALM report output format.

3 Making Entities Sharing Work

This chapter describes best practices for sharing entities in ALM.

Roles and Responsibilities

Similar to the other entities in ALM, libraries and baselines have group permissions, for example, can create and can import. When importing a library, the importing user must have permission to create the relevant entities in the modules, for example, new requirement and new test.

Since the import and synchronization processes may result in data corruption or loss if not managed properly, permissions should be carefully planned based on a person's role in this process. The following are examples of useful roles that were observed at ALM customer sites:

- Library Administrator (LibAdm)

  The Library Administrator is responsible for the library and its contents. They handle synchronizations, provide entities collision reports, and notify project managers about library usage by other project managers.

- Implementation Project Manager (ImpPM)

  The Project Manager is responsible for an implementation project, selects and imports libraries, defines milestones, and assigns reconciliations tasks to subject matter experts working on the implementation project.

- Repository Project Manager (RepPM)

  The Repository Project Manager is responsible for a repository/rollout project, selects and imports libraries, assigns reconciliations tasks to subject matter experts working on the implementation/repository project, and provides stable baselines for import.

- Implementation Project Contributor (ImpPC)

  People acting in this capacity are usually business analysts, test engineers, QA managers and the like. They have access to and control over certain information, like a specific application's requirements or test sets.

- Repository Project Contributor (RepPC)

  Like the previous role, people in this role handle the portion of the library tree that the LibAdm assigned to them.

- Library Reconciliation Board (LRB)

  This is a board of Library Administrators, Implementation Project Managers, and Repository Project Managers. They plan reconciliation activities, and analyze entities collision reports based on their expertise and input from subject matter experts who are Implementation Project Contributors and Repository Project Contributors.

  The user who performs the import operation, such as LibAdm, ImpPM and RepPM, should have permissions for both the origin project and the destination project. The permissions for both projects do not need to be the same. For example, a user can be a super-admin in the origin project, and have limited privileges in the destination project, such as create and update permissions.

  Libraries in the group permission screen have a data hiding filter. This filter can be used to hide certain libraries from particular users, such as ImpPC and RepPC. To define specific criteria for

the data hiding filter, use user-defined fields for libraries.

Remember that the above definitions are roles and responsibilities, not people. One person could fulfill multiple roles within the SDLC organization structure.

Development Scenarios

Libraries and baselines are powerful tools that can be used across the SDLC. We **recommend** following these rules when using these tools in your organization:

- Use a naming convention and/or custom attributes for libraries to enable the correct identification of their type, scope, status.
  - Type: Compliancy, Entities, etc.
  - Scope: description of the contents
  - Status: new, ready, under maintenance, etc.
- Use a naming convention and/or custom attributes for baselines to enable the correct identification of their type.
  - Type: after synchronization, after reconciliation, before synchronization, etc.

It is possible to enforce naming convention by using workflow code.

Part of the LibAdm responsibility is to inform ImpPMs about content changes so they can choose the proper library and baseline. In some cases, when there are a large number of entities and widespread use in child projects, consider implementing an automatic notification feature, such as automatic email to ImpPMs, for changes in the rollout project.

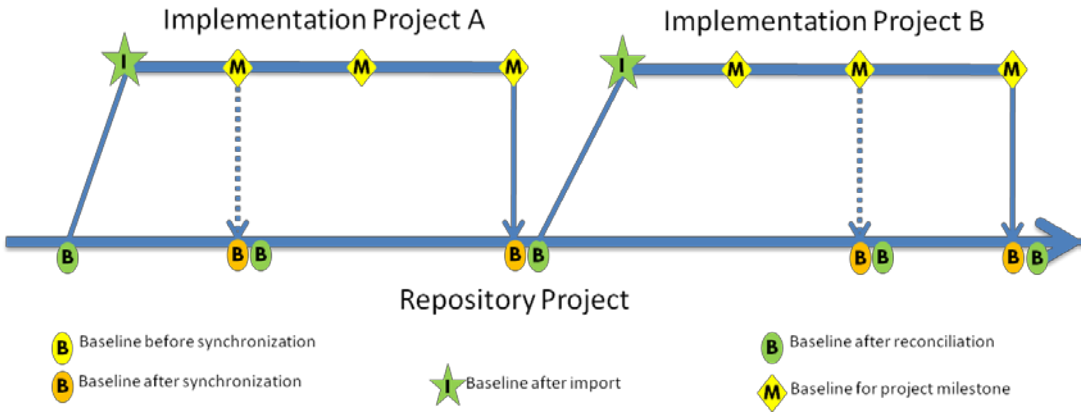
Before sharing entities, make sure these conditions are met:

- Repository/rollout projects are available and contain all relevant libraries.
- Implementation projects have already been created using the cross- project customization feature and libraries have been imported as part of project setup.

There are two common development scenarios in which entity sharing is an integral part of the process – sequential development and parallel development. ALM natively supports both scenarios and provides the means to achieve maximum productivity by using this feature.

Sequential Development

This is the most popular way to develop and test software. Tasks come one after the other and synchronization is performed at certain, predefined points of time.



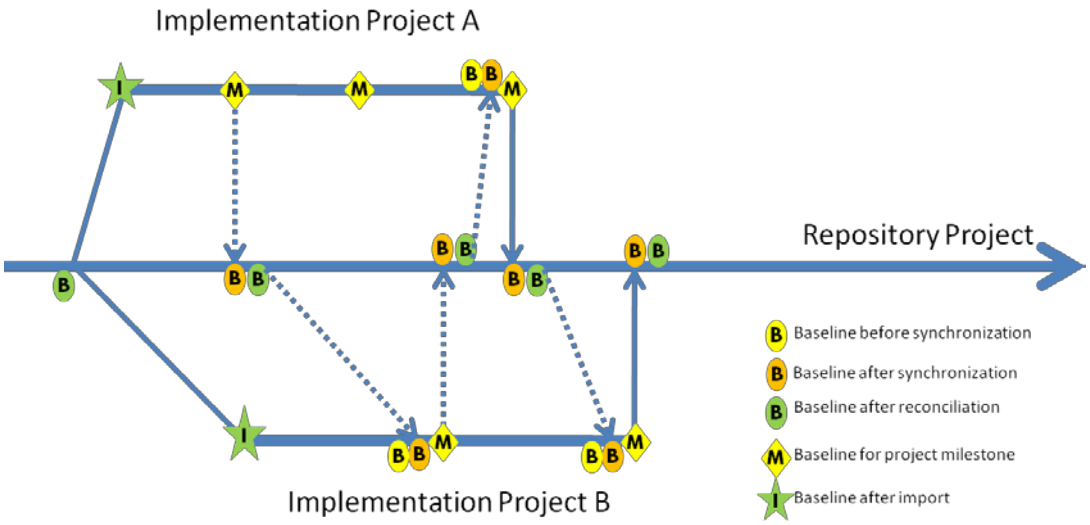
Here is a typical flow of events in a sequential development project:

- ImpPM reaches a project milestone that requires the synchronization of imported libraries (from implementation project into repository project)
- ImpPM asks ImpPCs to commit a stable revision of assigned entities (and/or group of entities)
- ImpPM creates the baseline for project milestone
- LRB compares the implementation project baseline and the repository project baseline to assess the impact
- LRB plans for reconciliation activities and assigns them to ImpPCs and/or RepPCs
- LibAdm performs library synchronization for the repository project (gets the latest/stable baseline from repository project)
- Baseline after synchronization is automatically created
- ImpPCs and/or RepPCs perform reconciliation tasks
- LibAdm creates a baseline after reconciliation

In the case of sequential development, more than two projects may synchronize with a single repository/rollout project.

Parallel Development

This way of development is primarily used by agile teams to speed up delivery.



Here is the typical flow of events in a parallel development project:

- ImpPM of Project A reaches project milestone that requires the synchronization of imported libraries (from implementation project to repository project)
- ImpPM of Project A asks his/her ImpPCs to commit a stable revision of assigned entities (and/or group of entities)
- ImpPM of Project A creates the baseline for project milestone
- LRB compares the implementation project baseline and the repository project baseline to assess the impact.
- LRB plans for reconciliation activities and assigns them to ImpPCs of Project A and/or RepPCs
- LibAdm performs library synchronization for the repository project (gets the latest/stable baseline from repository project)
- Baseline is automatically created after synchronization
- ImpPCs of Project A and/or RepPCs perform reconciliation tasks
- LibAdm creates a baseline after reconciliation
- ImpPM of Project B reaches project milestone that requires the synchronization of imported libraries (from implementation project to repository project)
- ImpPM of Project B asks ImpPCs of his/her project to commit a stable revision of assigned entities (and/or group of entities)
- ImpPM of Project B creates a baseline before synchronization
- ImpPM of Project B performs library synchronization for the implementation project, i.e., gets latest/stable baseline from repository project (from repository project to implementation project)
- Baseline is automatically created after synchronization
- ImpPCs of Project B performs reconciliation tasks
- ImpPM of Project B creates baseline for project milestone
- LibAdm performs library synchronization for the repository project, i.e. gets latest/stable baseline from implementation project (from implementation project to repository project)

- Baseline is automatically created after synchronization
- ImpPCs of Project B and/or RepPCs perform reconciliation tasks
- LibAdm creates a baseline after reconciliation

Due to the complexity and multiple meeting points, parallel development is typically used between no more than two projects/development teams.

Connection to Source Code Control

Regardless of the chosen development method – be it sequential or parallel, in most of the cases the development teams keep source code in a shared repository managed by Source Code Control (SCC) system. This is vital in handling the actual source and objects associated with the coding effort.

As with Application Lifecycle Management, managing source code follows a similar pattern with either a sequential or parallel development. Branching and merging, creating baselines/check point and many other lifecycle concepts directly apply to SCC tools.

ALM can plug into the leading development environments such as Eclipse and Microsoft Visual Studio and source code control systems such as Subversion. ALM Connector, developed by TaskTop, integrates its task-focused interface technology with ALM, resulting in improved developer productivity and interoperability with integrations with primary commercial and open source ALM platforms.

Even if there is no automatic connectivity between your SCC and ALM, it is still valuable to track the SCC baselines and to relate them to the baselines created in ALM using certain procedures and naming conventions.

We **recommend** that every time a baseline in ALM is created, a notation of the SCC tool baseline label or numbering pattern is written in the library baseline comments. For example, the baseline comments in ALM may contain "Subversion Release 4.3 beta" or at least "Checkpoint 1.7". This should allow for an easy correlation of the two related entities. On the other hand, a reference could also be placed in the SCC checkpoint comments area to reference its corresponding baseline in ALM.

Reconciliation Tasks

As mentioned in previous *sections*, various data conflicts may arise during the synchronization process. To ensure data consistency, follow these guidelines when performing data reconciliation tasks in the repository project:

- Delete an entity that has been created in repository project
  - ImpPC is required to delete the entity
- Restore an entity that has been deleted in repository
  - ImpPC is required to move a deleted entity from the "SYNCH_OBSOLETE_*" folder to its original node (folder or entity)
- Keep entity in "Baseline after synchronization" (latest revision)
  - Nothing has to be done

- Keep entity in latest “Baseline before synchronization” (previous revision)
  - ImpPC is required to check-out a previous revision of the entity and perform an immediate check-in
- Merge entity from both baselines (create a new revision)
  - ImpPC is required to check-out a revision of the entity that represents the best starting point for the reconciliation, (i.e., the revision containing most of the information that needs to be kept), apply the changes approved by the LRB, and check-in the entity.

4 Conclusions

The wave of application modernization has clearly started showing its power. Driven by new technologies and the quest for simplification and decreased costs, modernization touches almost all aspects of IT. It turns local, dedicated teams into virtual, distributed ones. It reshapes applications from monolithic blocs of software to composite "systems of systems." It enriches user experience and company brand via Web 2.0 and rich Internet applications. It changes release management from single launches to multi-application "release trains." To truly enable business change, IT managers must examine their approach to planning, developing, deploying, and operating software applications.

Since applications are no longer encapsulated, projects are no longer encapsulated. Functionality, performance, and security decisions made within a project affect IT services throughout the enterprise. Sharing entities between projects and thus allowing collaboration between various stakeholders in IT helps keep up the pace of the ever-changing essence of today's applications. ALM provides all of the necessary means for sharing artifacts. This is particularly vital in larger enterprises where team collocation is not always viable.