Version Control Best Practices for Enterprise Architect

www.sparxsystems.com

All Material © Sparx Systems, 2010

Table of Contents

Glossary....3
Introduction....4
    Why Should I consider version controlled models?....4
    What does 'version control' mean in Enterprise Architect?....4
Team Deployment: Centralized or Distributed....5
Scenario 1: Centralized Team....6
    Recommended process for version controlling models....7
    Recommended process for rolling back (undoing) changes....8
Scenario 2: Distributed Team using local models....8
    Recommended process for version controlling models....9
    Managing cross-Package dependencies....9
    Recommended processes for submitting changes safely....10
    Recommended process for rolling back (undoing) changes....14
Scenario 3: Multiple Site Locations....15
Appendix A: Enterprise Architect meta-data that is not stored in the Version Control Repository.. 16
Appendix B: Built-in Collaboration and Change Management Tools....16
Appendix C: Applying Version Control to Packages....17

Glossary

Baseline (Model Baseline): In Enterprise Architect, a Baseline refers to a snapshot of a Package at a particular point in time. The snapshot is stored in the Model Repository as compressed XMI and forms the basis of Enterprise Architect's Compare and Merge functionality.

Check-in: The process of submitting your changes to the Version Control Repository. In Enterprise Architect, you execute this command on a Package that you have checked out. This then updates the Version Control Repository with your changes and releases your editing lock on that Package.

Check-out: The process of retrieving the latest version of a file from the Version Control Repository. Executing this command from Enterprise Architect will overwrite the selected Package with the latest version and lock that Package for your exclusive editing.

Commit: Submit your changes to the Version Control Repository without releasing your editing lock on the related file(s). In Enterprise Architect, this is equivalent to executing the “Put Latest” command.

DBMS: Database Management System. Enterprise Architect supports several relational DBMS products to host the Model Repository. A DBMS is commonly used when the Model Repository is accessed by several concurrent users.

EAP: Enterprise Architect Project. The abbreviation usually refers to the file-based Model Repository, hence 'EAP file.'

Get Latest: A command executed from within Enterprise Architect on a selected Package. Updates that Package with the latest information from the Version Control Repository, without performing a Check-out. **Get All Latest** updates all version controlled Packages in the project.

Model Repository: Enterprise Architect's storage mechanism for model information. If multiple users concurrently edit the same model, the repository is usually DBMS-based. In distributed team environments where concurrent model access is not possible, local EAP files are usually deployed instead.

Enterprise Architect can use your Version Control System to transfer information between the Model Repository and a Version Control Repository to propagate changes amongst team members.

Package: The primary organizational construct in Enterprise Architect models, roughly equivalent to a 'folder' in the file system. A Package may be a model's Root node, a View or a sub-Package.

Put Latest: A command executed from within Enterprise Architect on a selected Package that you have checked out. Updates the Version Control Repository with your changes to that Package, while retaining its 'Checked-out' status. (Equivalent to checking a Package in and immediately checking it back out again.)

Version Control Configuration: In Enterprise Architect, a Version Control Configuration records connection settings for the Version Control Repository and the path to your local Working Copy.

Version Control Repository: The storage mechanism used by the Version Control System to store revisions – specifically, model revisions.

Version Control System: The third-party product that manages revisions of your model data. Enterprise Architect supports several Version Control Systems (such as CVS, Subversion and SCC-compliant products) and provides the user interface needed to move data between the Model Repository and the Version Control Repository.

Working Copy: The set of files on your local machine that you have retrieved from the Version Control Repository. Enterprise Architect uses your working copy files to update the Model and Version Control Repositories.

XMI: XML Meta-data Interchange. An open standard file format that enables the interchange of model information between tools. Enterprise Architect uses this file format to serialize a Package's model information to facilitate storage in the Version Control Repository. When you apply version control to a Package in Enterprise Architect, one XMI file is created for that Package (and its contained elements) and added to the Version Control Repository.

Introduction

This document explains how version control concepts apply to Sparx Systems Enterprise Architect and suggests best practices for establishing version control in shared and distributed modeling environments. The information presented here is intended for Enterprise Architect users who:

- Want to know whether applying version control to their model could be beneficial and what alternatives are available or;
- Require best practice advice for specific Enterprise Architect deployment scenarios or;
- Already have experience with version controlled models and wish to improve how these are managed.

Why Should I consider version controlled models?

In general, benefits of using version control systems include: increasing the potential for parallel and distributed work, improved ability to track and merge changes over time and automating management of revision history. Following are some specific benefits that can be realized by applying version control to a modeling environment. These are most relevant when models are shared by multiple editors, who may also be geographically dispersed:

- Supports globally distributed model editing by providing a convenient and effective way of replicating models.
- Facilitates collaboration across multiple projects through re-use of common model data.
- Improves performance for widely dispersed teams over slow networks by allowing local storage of models, with global propagation of changes only.
- Promotes orderly changes over chaotic changes and helps to minimize disruptions by separating "work in progress" from "finished" work.
- Helps automate communication within a team by coordinating 'edit' access to controlled information, thereby preventing accidental modification.
- Helps maintain successive revisions of work-to-date with the ability to "undo" changes that are not required, "roll back" to the last "good" version to undo a mistake, or recover from accidental deletions or changes.
- Maintains a work history and audit trail of changes to a model, helping to answer 'Who changed what and when?'

What does 'version control' mean in Enterprise Architect?

Enterprise Architect applies version control to Packages within a model. A Package is the primary organizational construct for UML models. Any Package may have version control applied to it - whether it is the model's Root node, a View or a sub-Package (see Figure 1).

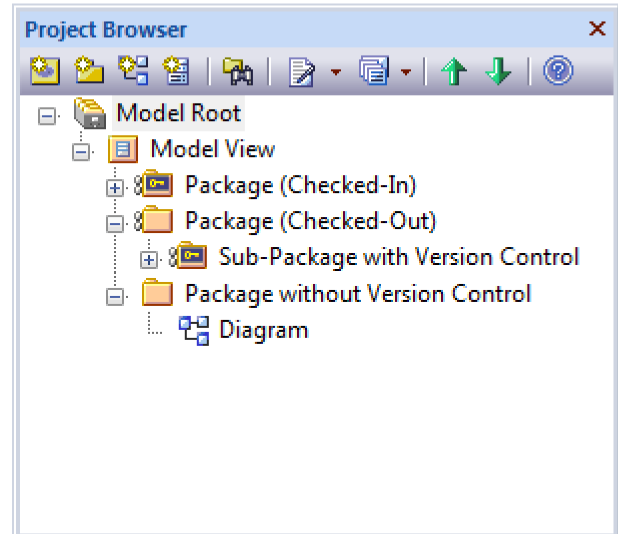

Figure 1: Version Control at the Root, View, Package or Sub-Package

Enterprise Architect supports two primary ways of version controlling Packages in a model:

1. **Model Baselines**: This built-in capability, stores point-in-time snapshots of a Package in the model repository itself. The Model Baselines concept also forms the basis of Enterprise Architect's 'compare and merge' facility.

   When deciding on approaches to version control, strong consideration should be given to the use of Model Baselines, especially where the need is essentially to maintain a revision history for purposes of comparison, merge and roll-back.

2. **Version Control Integration**: Enterprise Architect supports integration with third-party version control systems, allowing users to store Package revisions in their preferred tool. Version control tools supported by Enterprise Architect include Subversion, CVS, Microsoft's TFS and SCC compatible tools, such as Accurev and Visual Source Safe.

   For large, distributed teams it is often necessary to use such a dedicated version control system to manage broad-scale replication and sharing of model data.

This document focuses on the latter approach – achieving version control with third-party integrated tools. Readers who are interested in related change management features and simple alternatives, should refer to Appendix B: Built-in Collaboration and Change Management Tools

Figure 2 provides a high-level schematic of the relationship between Enterprise Architect and the external version control system. Note that versions are stored and retrieved as XMI files. XMI is used to serialize Package information to create a point-in-time snapshot. Enterprise Architect enforces that only a single user edits a given Package at once. This approach represents a "Lock-Modify-Unlock" solution, which helps to avoid revision conflicts – especially useful as XMI files are considered binary artifacts that cannot be merged directly via the Version Control System.

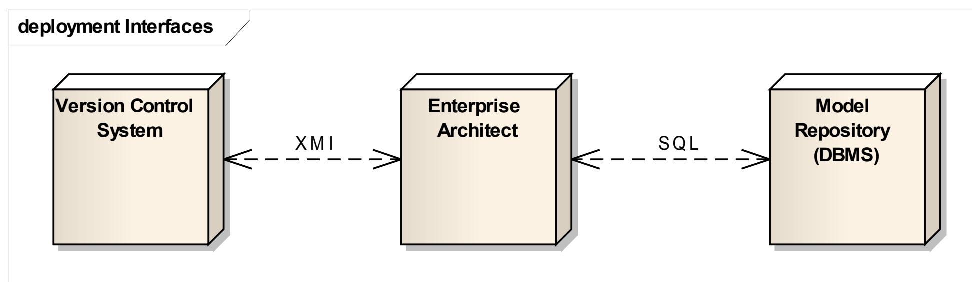

Figure 2: Enterprise Architect exchanges model data with the version control systems using XMI files

The rest of this document is designed to help you identify your deployment scenario for Enterprise Architect and to suggest suitable version control approaches. For each approach, we also highlight specific challenges that need to be addressed. Instead of providing detailed "how-to" steps for setting up your version control system with Enterprise Architect, we provide relevant references to the Enterprise Architect User Guide for further information.

Team Deployment: Centralized or Distributed

The primary consideration in deciding which approach to take in version controlling your model is how the model editors, or authors, are distributed:

- *Is each member of the team in the same central location and connected via a high-speed network?*
- *Are the editors likely to work remotely and independently, perhaps disconnected from a shared network for extended periods of time?*
- *Are there several key locations across the globe where the model will be shared and edited?*

Answers to these questions will determine how the Enterprise Architect model itself is shared[1] and consequently how version control is applied. In the following sections we describe how to apply version control in some common scenarios. Listed below are the scenarios we will consider in detail:

1. **Centralized Team:** All editors are connected via a high-speed network and can therefore share the same physical model hosted in a Database Management System (DBMS).

2. **Distributed Team:** Editors are rarely connected to the same network. They may have to contribute to the model offline, and therefore require a local copy of the model on their own machine.

3. **Multiple Site Locations:** Several geographically dispersed sites contribute to the same model. There is no high-speed connection between sites. Teams at each site share access to a local copy of the model.

Note: Future updates to this whitepaper will also address the concept of distributing 'model libraries' – models that serve as self-contained components to be reused across several projects by any number of 'consumers'.

1 The white paper Deployment of Enterprise Architect addresses how to deploy the modeling tool itself in various team-based scenarios.

Scenario 1: Centralized Team

In this scenario, we have the benefit of all team members being connected by high-speed network infrastructure. If our team exceeds 5 members who concurrently access the model, we can no longer safely access a single EAP file located on a network share. Instead, it is recommended to use a dedicated DBMS to host our model. The advantage of using a shared DBMS in this situation (as opposed to local copies of the model) is that all team members view and edit the current-state model, without the need for a separate synchronization process required to "get the latest."

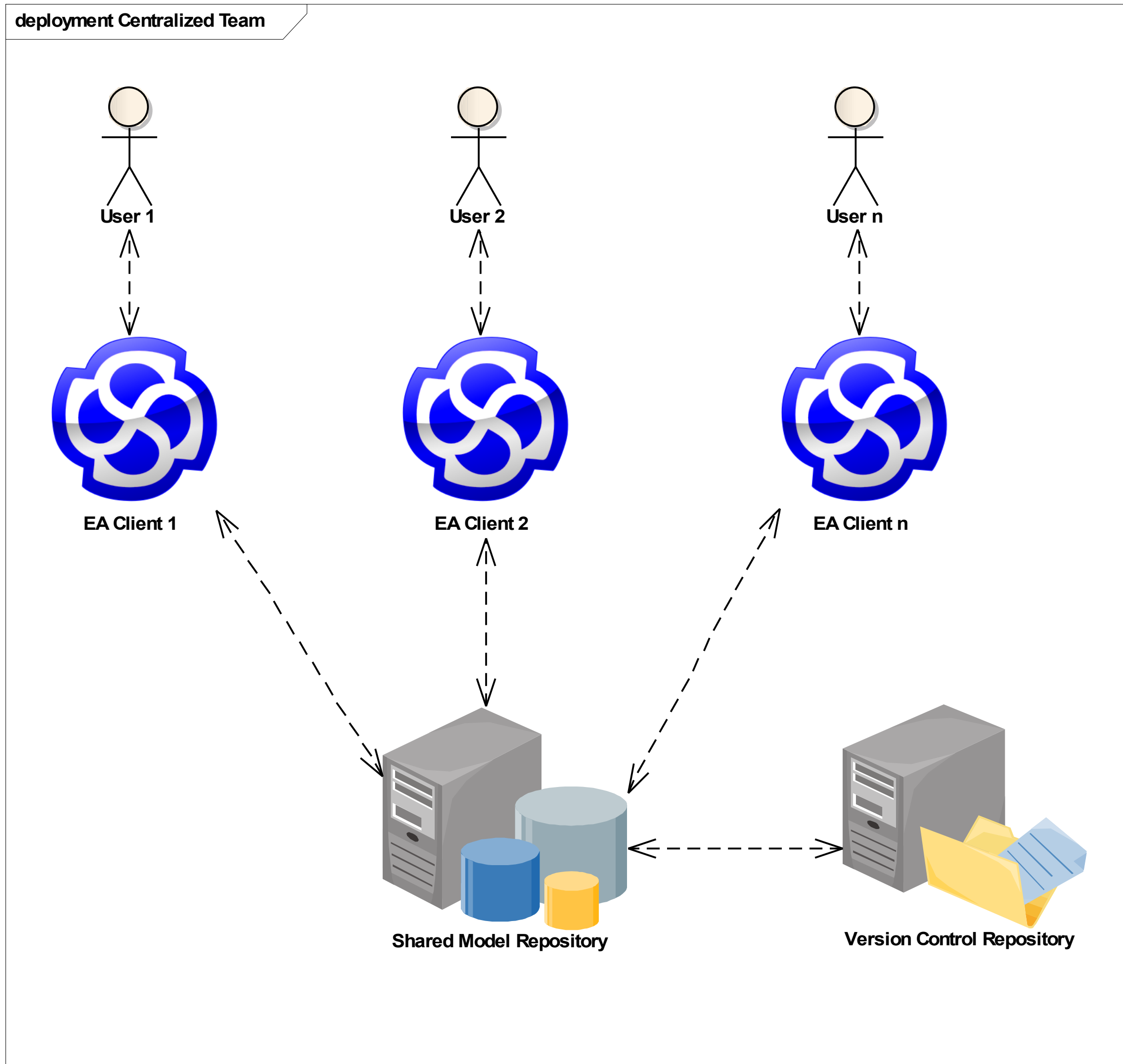

Figure 3: A Centralized Team connects to a common DBMS to edit the model

Therefore, the role of the version control repository in this situation is not to provide a mechanism for distributing model information – a function that is already achieved by the DBMS. Instead, the version control repository helps to manage revisions and allow for roll-back of changes.

Following are some common questions that arise when administering version control in such a scenario:

- At what level of granularity do I apply version control – Model level, Package or sub-Package level etc?
- How do I stop one team member from overwriting the work of another?
- If someone makes a mistake, how can the previous state of the Package be safely restored?

Answers to these questions are provided in the following Recommended Processes and Best Practices.

Recommended process for version controlling models

1. **Setup a DBMS repository:**
   (a) Establish a dedicated DBMS server supported by Enterprise Architect and accessible to all team members.
   (b) Run the SQL scripts for Enterprise Architect's database schema and then transfer any existing starter models. (Detailed instructions for using the scripts are available from the above link).
   (c) Optionally, enable User Security on the Enterprise Architect model. This allows you to control who has access to particular features of Enterprise Architect in this model.

2. **Setup a version control repository:**
   (a) Install the server component of your preferred, supported Version Control application, and ensure team members have the appropriate client software installed.
   (b) Create a version control repository with an empty project that will be used for Enterprise Architect.

3. **Setup version control clients:**
   (a) Use a client machine to check-out a working copy of the empty project into a local folder.
   (b) In Enterprise Architect, define a version control configuration, which accesses the working copy files. The process for defining a version control configuration varies according to your version control system. Each user that subsequently accesses the DBMS model repository for the first time will be prompted to configure their local version control settings. They will reuse the configuration you have established for this model and specify the path to their own working copy files and version control client executable.

4. **Control Packages:**
   (a) In Enterprise Architect, apply version control to individual Packages. Refer to Appendix C for details.

After implementing the above setup procedure, users will checkout Packages in Enterprise Architect under their version control user account. The check-out process invoked within Enterprise Architect locks the Package for exclusive editing by that user.

In this scenario, there is generally no need to perform the “Get All Latest” or “Get Latest” because the model repository always contains the latest model information. (The stored revisions in the Version Control Repository originated from the model repository.)

Best Practice 1: In a Centralized Team model, apply version control to all Packages in the model hierarchy, which includes sub-Packages and Root nodes, to maximize potential for parallel work.

Best Practice 2: Use Enterprise Architect's role-based (user) security to restrict functionality through user and group permissions and to enable Work flow scripting. For example, you might establish a security group that has permission to administer version controlled Packages, enabling such permissions as 'Configure Version Control' and 'Configure Packages'.

By selectively adding administrative users into a group, you can better manage which Packages are added to (or removed from) version control, according to your guidelines. Furthermore you can restrict which users can checkout Packages by removing the 'Use Version Control' permission, hence restricting users to read-only access of version controlled Packages. You can also use your version control system's access permissions on specific folders to restrict read/write access of Packages stored in those folders.

Note: It is not recommended to use role-based security as a means of applying user and group locks to version controlled Packages. Package-level locking is already achieved via the version control system.

Best Practice 3: It is useful to periodically archive the entire shared model, both for back-up purposes and to allow viewing the model when disconnected from the network. You can achieve this via Enterprise Architect's capability to transfer a DBMS project to EAP format (use the DBMS to .EAP option). This model transfer should be done in addition to your standard DBMS back-up procedure, not as a substitute.

Best Practice 4: Always include meaningful comments when checking-in a Package (Enterprise Architect prompts you to add a comment via your version control system's check-in procedure.) A high-level comment that meaningfully describes the nature of your changes is helpful when later reviewing the change history. Such comments are also useful indicators of the last 'good' state, should you need to restore a Package to a prior revision in the case of error.

Recommended process for rolling back (undoing) changes

1. Retrieve a prior revision – the last known 'good' state – of the Package from version control. Use the Package's File History in Enterprise Architect to select the particular revision and import it into the model.
2. Create a Baseline of the retrieved Package.
3. Check-out the Package in the model. This unlocks the Package for editing, but in doing so, retrieves and imports the 'head revision' of the Package from Version Control.
4. Using the Baseline created earlier, 'Restore to Baseline'.
5. Check-in the (restored) Package.
6. Optionally, delete the baseline created in step 2.

Scenario 2: Distributed Team using local models

In this scenario we have multiple individuals editing the model, but without being connected via a shared Model Repository. Instead, each editor maintains a local copy of the model as an EAP file and periodically updates their copy from a shared Version Control Repository. This approach facilitates broad-scale replication of the model without the need for administering a DBMS.

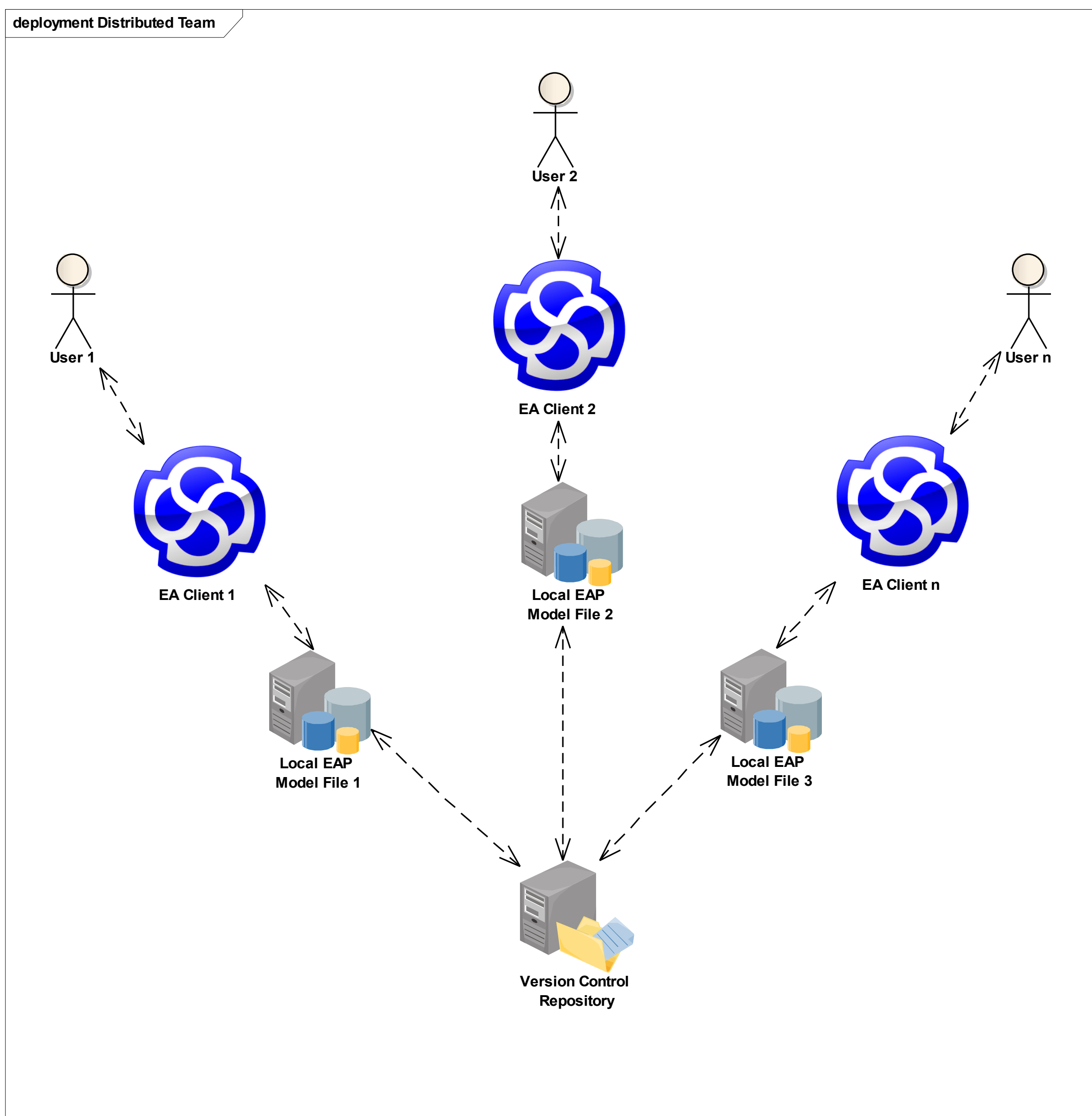

Figure 4: A Distributed Team updates local EAP files from a common version control repository

Recommended process for version controlling models

1. **Set up an initial version controlled Enterprise Architect model:**
   a) Create a version control repository.
   b) Using your version control system, create an entry in your version control repository that will be used to store your Enterprise Architect Package files.
   c) Create a working copy in a local folder (for example, in CVS and Subversion this requires checking-out the entry created in the previous step).
   d) Have one user create an Enterprise Architect project (EAP) file and define a version control configuration. The configuration provides access to the working copy files within Enterprise Architect.
   e) In Enterprise Architect, apply version control to individual Packages. Refer to Appendix C for details.

2. **Distribute the model to the rest of the team:**
   a) Once the model is created with version control applied, distribute the EAP file to the rest of the team.
   b) Every team member that will access the version controlled Packages from Enterprise Architect must create a working copy as in step 1c.
   c) As team members subsequently open the model for the first time, they will be prompted by Enterprise Architect to complete the definition for all version control configurations that are used by the model. This entails simply specifying the path to the local working copy files and saving the definition.
   d) The model is now connected to version control and ready for use.

Best Practice 5: As with a Centralized Team, you can maximize parallel work by applying version control to all Packages in the model hierarchy, including sub-Packages and Root nodes. If you have carefully managed cross-Package dependencies (described in the following section) this is still appropriate. In a distributed environment, this approach has an additional performance benefit, as each user will transfer smaller XMI files between the version control and model repositories when committing and retrieving the latest changes.

The editing process can be simplified, however, by reducing the dependencies between *version controlled Packages.* To achieve this, you may decide not to apply version control to lower levels in the model hierarchy. There is a trade-off between reducing the potential for missing cross-Package dependencies and performance and parallel work.

Best Practice 6: Assign one team member as a 'Model Manager', responsible for maintaining a 'master' EAP file. This EAP file will not be used for day-to-day work. Rather its purpose is to provide a starting point for new team members. The copy of the model received by the new team member should not contain any Packages already marked as checked out to another user. The master EAP file should be updated from the version control repository using the 'Get All Latest' command in Enterprise Architect and any new top-level Packages created must be added to the model using the 'Get Package' command. Define a process for alerting the Model Manager of any new Packages added to version control.

By using copies of a 'master' EAP file and allowing Enterprise Architect to prompt the user to complete definitions for required version control configurations, you ensure consistent spelling of configuration IDs across each copy.

Best Practice 7: In a distributed team environment, where team members edit local EAP files, there is no automatic update of Enterprise Architect's role-based security information, as it is not stored in the version control repository. Therefore, it is not recommended to apply role-based security in this scenario.

Managing cross-Package dependencies

As Packages can be controlled independently of each other, you can split a model (intentionally or otherwise) such that an element at one end of a relationship is not present in the particular copy of the model you are editing. This may be intended to restrict the scope of model information required by an editor of the model. Such a scenario however, can lead to potential loss of model information. Consider the model sample shown in Figure 5.

Classes **Parent** and **Child** are defined in separately version controlled Packages and have an inheritance relationship between them. **Child** is defined in **Package A**, whereas **Parent** is defined in **Package B**. This scenario demonstrates one example of a *cross-Package dependency* between version controlled Packages.

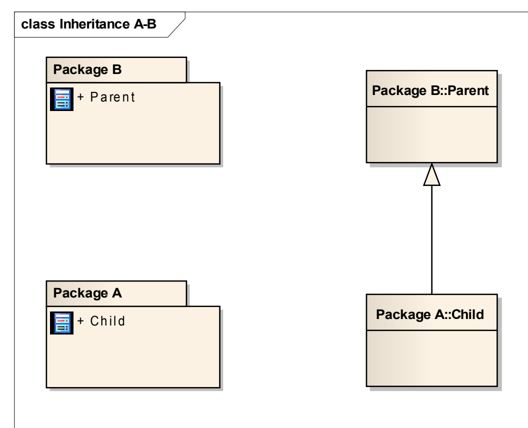

Figure 5: Parent and Child defined in separate Packages

The model hierarchy is shown in Enterprise Architect's Project Browser in Figure 6.

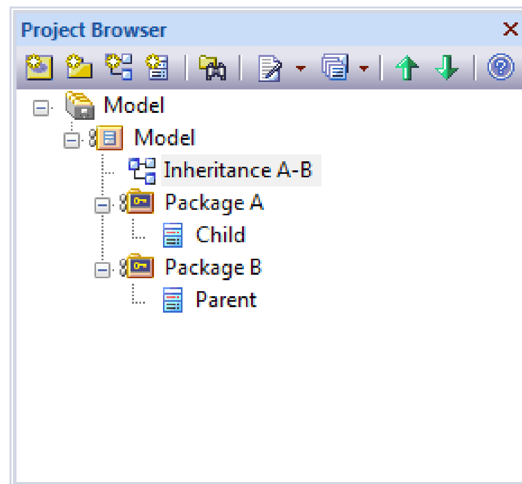

Figure 6: Related Packages independently version controlled

When both Packages exist in the model, the implicit dependency between Packages A and B is satisfied via the connected elements. Conversely, if we had only one of the Packages in our local model, the dependency is broken – one of the required elements would be missing. The immediate result would be that any diagrams referring to the relationship would be incomplete. Furthermore, if we checked out the Package to our incomplete model and committed a new revision, the resulting XMI would not include the relationship, thereby semantically altering the model.

In any non-trivial project, you will almost certainly model dependencies between version controlled Packages. So far, we have considered only one scenario where this occurs. In the subsections to follow, we consider other specific modeling situations that involve cross-Package dependencies and the recommended processes for safe editing.

Best Practice 8: Plan your Package dependencies in advance and maintain the known set of dependencies. This can be achieved using UML Package diagrams and explicitly modeling UML Dependency relationships between them. Maintain the Package diagrams in a copy of your 'master' EAP file, where all Packages are guaranteed to be available. Define modeling conventions and guidelines that prevent modelers introducing unplanned dependencies.

Enterprise Architect has built-in tools that help you identify how a given element relates to other elements in the model. Examples include the Traceability Window, the Relationships Window and the Relationship Matrix. You may also benefit from scripts such as the Package Dependency script provided via the Sparx Systems Enterprise Architect Community site, which helps you automatically create dependency diagrams.

Best Practice 9: Always work on the complete model. Much of the complexity associated with cross-Package dependencies can be avoided simply by ensuring your local EAP file contains a copy of the entire model, rather than a partial model. By starting with a copy of the 'master' EAP file, and regularly executing the “Get All Latest” command, you minimize the risk of submitting changes with missing dependencies. *It is strongly recommended that you run 'Get All Latest' prior to checking out any Package.*

Recommended processes for submitting changes safely

The following scenarios describe situations where updates to the model affect multiple version controlled Packages because of a dependency between them. The dependency may be explicit, such as a UML Inheritance relationship, or implicit, such as referring to a Classifier from a separately version controlled Package to specify an Attribute's type.

Modeling connectors between two version controlled Packages
Assume that we have two independently version controlled Packages, A and B, as modeled in Figures 5 and 6. Now assume we wish to perform the following editing operations between elements in the two Packages:

1. Create a connector from an element in Package A to Package B
2. Change the connector's target element to a different element
3. Reverse the connector's direction, change the connector's type and model bi-directionality
4. Delete the connector

For each of the above model edits, we define a recommended process for updating the corresponding Packages. Using the recommended process will help to ensure all other model editors can receive our updates correctly. That is, the edits we see in our local copy of the model, get reflected in the corresponding XMI files used to update all other models connected to the same version control repository.

1. Create a connector from an element in Package A to Package B

We want to model the situation illustrated in Figure 7. We can consider that class **Child** effectively 'owns' this relationship, as the inheritance does not semantically alter **Parent**. So we need only check-out **Package A** to make the change.

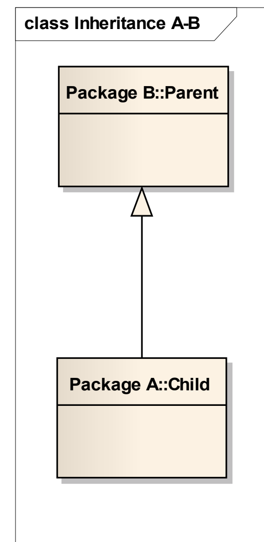

Figure 7: Create Connector

The process for creating any such uni-directional connector is as follows:

i. Use “Get All Latest” to ensure you have the whole model (Best Practice 9)
ii. Check-out Package A
iii. Create the relationship
iv. Check-in Package A

Note: If you create the relationship using a diagram in another Package, it would also need to be checked-out and checked-in along with Package A.

2. Change the connector's target element to a different element

This edit is essentially identical to our previous edit because we do not semantically alter either the original target element or the new target element by re-routing the connector. So we can apply the same process, replacing step 3 with the change of target element.

3. Reverse the connector's direction, change the connector's type and model bi-directionality

If we now wish to edit the same model to make the sequence of changes illustrated by Figures 8 – 10, we are making a change that semantically alters elements both in Package A and Package B. Therefore, we need to check-out both Packages to make the change and subsequently check-in both Packages to ensure the XMI files in the version control repository reflect our updated model.

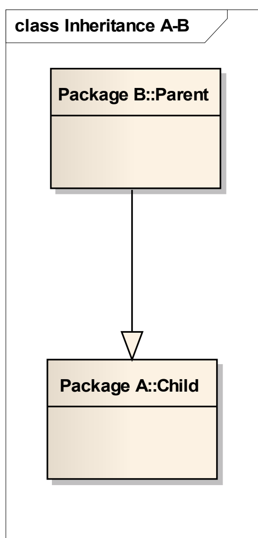

Figure 8: Reverse Direction

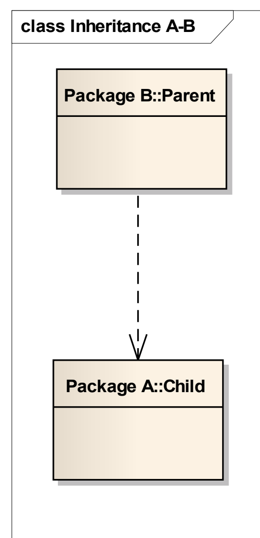

Figure 9: Change Type

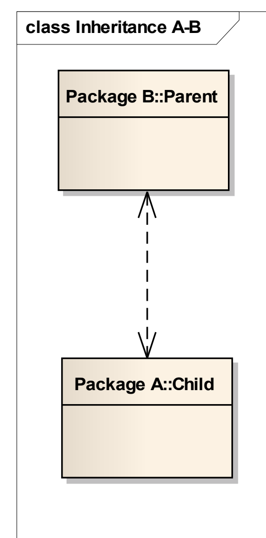

Figure 10: Make bidirectional

The recommended process for effecting these changes is as follows:

i. Use “Get All Latest” to ensure you have the whole model (Best Practice 9)
ii. Check-out Package A and Package B
iii. Modify the relationship
iv. Check-in Package A and Package B using “Check-in Branch” (Best Practice 10)

4. Delete the connector
Now assume we have our connector in its original state as modeled in Figure 7 and we wish to delete the connector. Currently, Enterprise Architect requires that we use the same process as above – ie. Checking-out both Packages. Although we do not semantically alter the Parent class by removing the inheritance relationship, information about the connector is stored in both Package's XMI. To ensure the connector is not later restored via the XMI for Package B (for example by running "Get Latest"), we must update both Packages.

Best Practice 10: 'Atomic Commits'. When checking-in a self-contained change that affects multiple Packages, use Enterprise Architect's 'Check-in Branch' command. This command allows you to commit all affected Packages at the same time – thus preventing other editors from checking out only part of your update and potentially losing related changes. It also allows you to use the same check-in comment for all Packages and to logically group the change set.

Best Practice 11: Commit small, self-contained changes regularly. If you keep several Packages checked-out over an extended period of time, you are likely to make numerous unrelated changes, increase the number of cross-Package dependencies affected by your changes and increase the complexity associated with rolling back changes.

Modeling classifier references between two version controlled Packages
Again, assume that we have two independently version controlled Packages, A and B. Assume that Package A has an element X, and Package B contains an element Y. Now we wish to model a reference from element X to element Y as a classifier. Examples of modeling situations where this occurs include:

- Specifying the type of attribute in element X
- Specifying the return type or parameter types of an operation in element X
- Specifying the classifier for element X, where X is an instance (or UML Object)

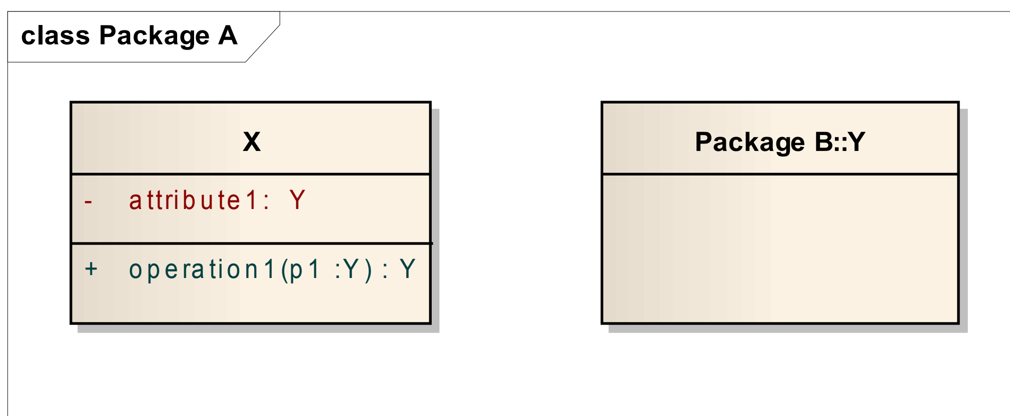

Figure 11: Class X refers to Class Y as a classifier for attribute and operation parameter types. Class Y is defined in a separately version controlled Package, Package B.

In these situations, we are again modeling a cross-Package dependency even though there is no explicit relationship being drawn between the elements. The recommended process for creating or updating these 'implicit' dependencies is:

i. Use "Get All Latest" to ensure you have the whole model (Best Practice 9)
ii. Check-out Package A and Package B
iii. Add, update or delete the classifier reference(s)
iv. Check-in Package A and Package B using "Check-in Branch" (Best Practice 10)

Given that we are not semantically altering Package B or its elements by setting classifier references in another Package, it is reasonable to ask: What is the need for checking-out Package B at all? In short, the XMI for both Packages contains the classifier reference information. If we only updated XMI for one Package (via the check-out/check-in process), there would be conflicting information between the respective XMI files. When subsequently executing a "Get All Latest" command, whichever Package is updated second will overwrite classifier information in the first Package.

It is useful, however, to have the classifier information in both XMI files. Consider, for example, populating a model from scratch. If we import Package A into an empty model first, Package B (and its classifier Y) do not yet exist, hence

the reference from element X to Y would be lost. Only by including the dependent references to Y in Package B's XMI do we resolve such 'dangling references' on subsequent import of Package B.

Note: Future releases of Enterprise Architect will obviate the need for checking out both the 'dependent' Package (Package A) and the 'target' Package (Package B), by rescanning the appropriate XMI files following 'Get All Latest'.

Moving an element between two version controlled Packages
Now assume we need to swap element X out of Package A, into Package B. In this case, we are clearly affecting the semantic model of both Packages. The update process is as follows:

i. Use "Get All Latest" to ensure you have the whole model (Best Practice 9)
ii. Check-out Package A and Package B (required by Enterprise Architect to allow the move)
iii. Move the Element X from Package A to Package B
iv. Check-in Package A and Package B using "Check-in Branch" (Best Practice 10)

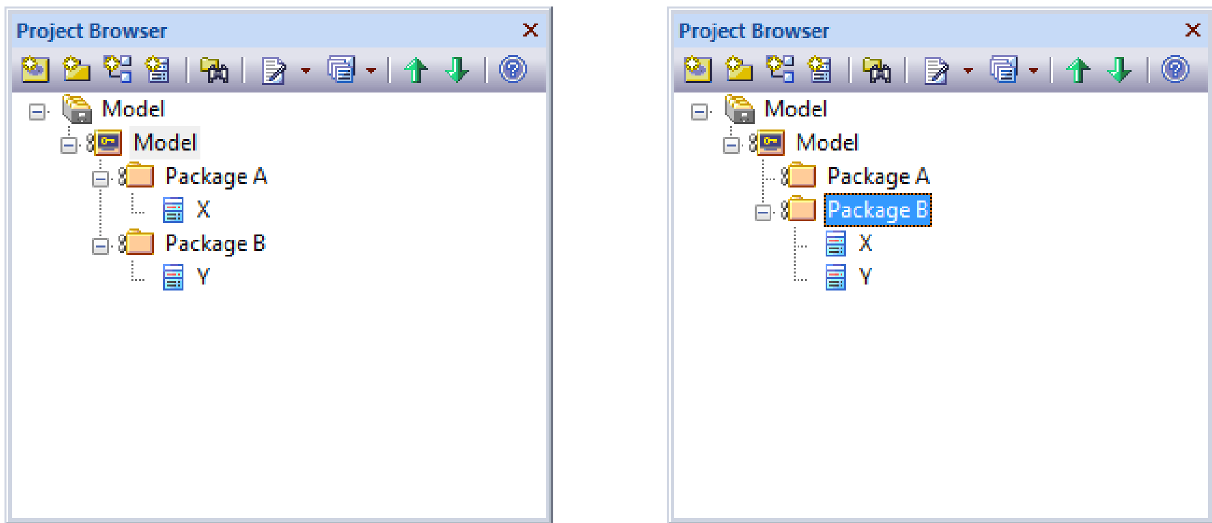

Figure 12: Moving an element semantically alters both Packages. Therefore we must check-out each Package.

A note on Sequence and Communication Diagrams
When creating sequence models, it is common practice to separate the classifiers, such as class elements in the 'Domain Model' or Actors in the 'Use Case' model, into different Packages from the sequence diagrams that use these elements. This is reasonable as it helps to better organize the model. An example model hierarchy is shown in Figure 13.

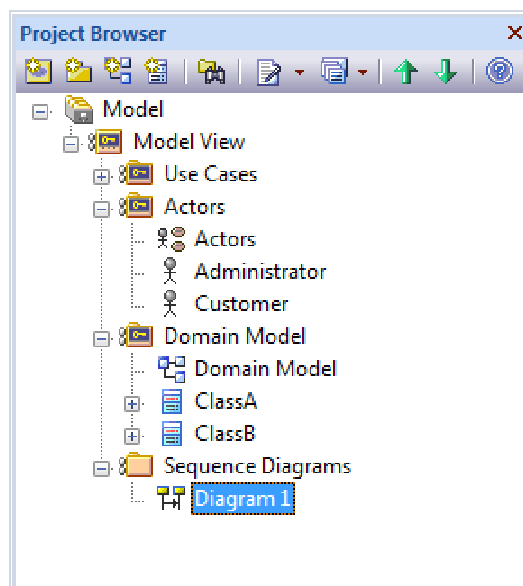

Figure 13: Use of 'Sequence Diagrams' Package

When using classifiers from these external Packages however, it is best to create instances from them on sequence diagrams. This is both semantically correct from a UML modeling perspective and also prevents potential loss of connector information on the diagram when round-tripping Packages via version control check-in/check-out. Sequence and communication diagram messages are only exported in the XMI for the Package that contains the diagram. If your instance elements and diagrams are in the same Package, all connector information is preserved during subsequent imports. Figure 14 illustrates the recommended modeling approach for a sequence diagram built from the above model.

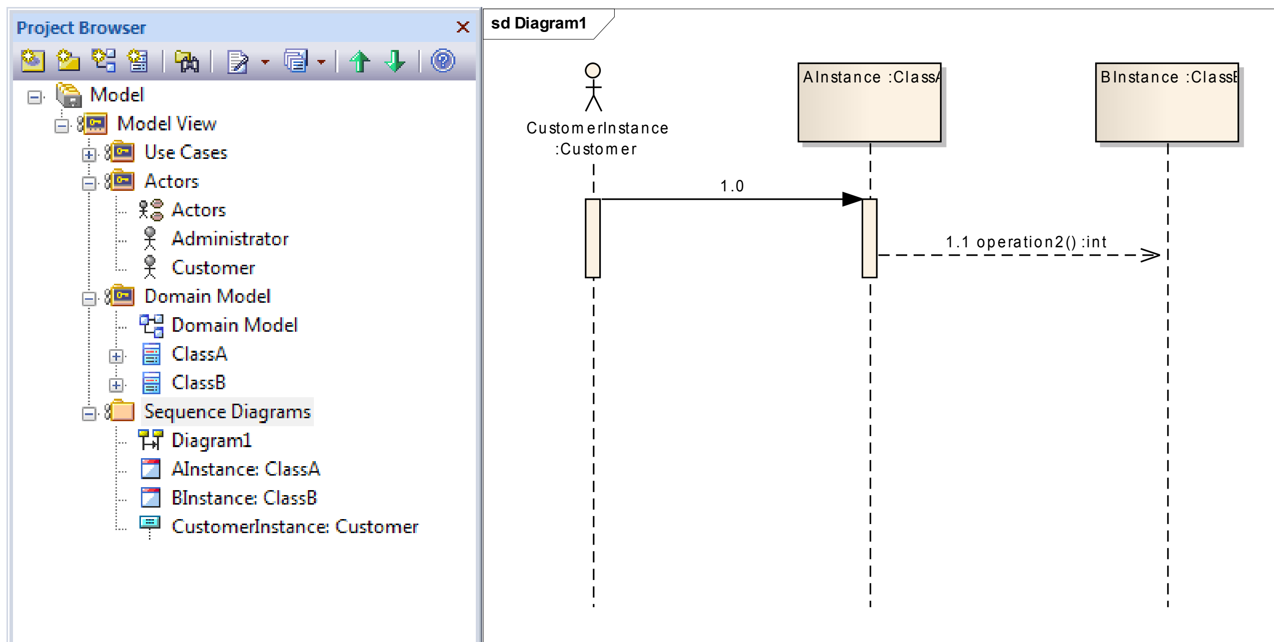

Figure 14: Using instances to reference classifiers is semantically correct and helps to preserve model integrity

Best Practice 12: When creating sequence and communication diagrams, use instances that refer to classifiers. (Do not use classifier elements directly). This is semantically correct and by keeping instance elements and diagrams in the same Package, ensures that messages are preserved during check-in and check-out.

Recommended process for rolling back (undoing) changes

Rolling back changes (or undoing mistakes) requires the same process described for Centralized Teams.

By way of a review, we list below the *Best Practices* introduced in this section in abbreviated form:

Best Practice 5: Apply version control to lower-level Packages to increase potential for parallel work. Balance this with the corresponding increased potential for dependencies between version controlled Packages.

Best Practice 6: Assign a 'Model Manager' responsible for maintaining a 'master' EAP file.

Best Practice 7: When using local EAP files to replicate the model, do not apply role-based security.

Best Practice 8: Plan your Package dependencies in advance and maintain the known set of dependencies.

Best Practice 9: Always work on the complete model. Use 'Get All Latest' prior to checking out any Package.

Best Practice 10: 'Atomic Commits'. When checking-in a change that affects multiple Packages, use 'Check-in Branch'.

Best Practice 11: Commit small, self-contained changes regularly.

Best Practice 12: On sequence and communication diagrams, use instances that refer to classifiers, thus keeping the instances and diagrams in the same Package.

Scenario 3: Multiple Site Locations

It is becoming increasingly common for large corporations to share model information across multiple, geographically dispersed development sites. The challenge then is to keep each site updated with the latest model information. While DBMS-level replication between sites may be possible using synchronization tools, leveraging version controlled Packages provides a simple, effective alternative. The situation represents a combination of the deployments defined in Scenarios 1 and 2. Each site may still benefit from a model repository hosted on a local DBMS, or EAP files may be used by individual editors, as shown in Figure 15.

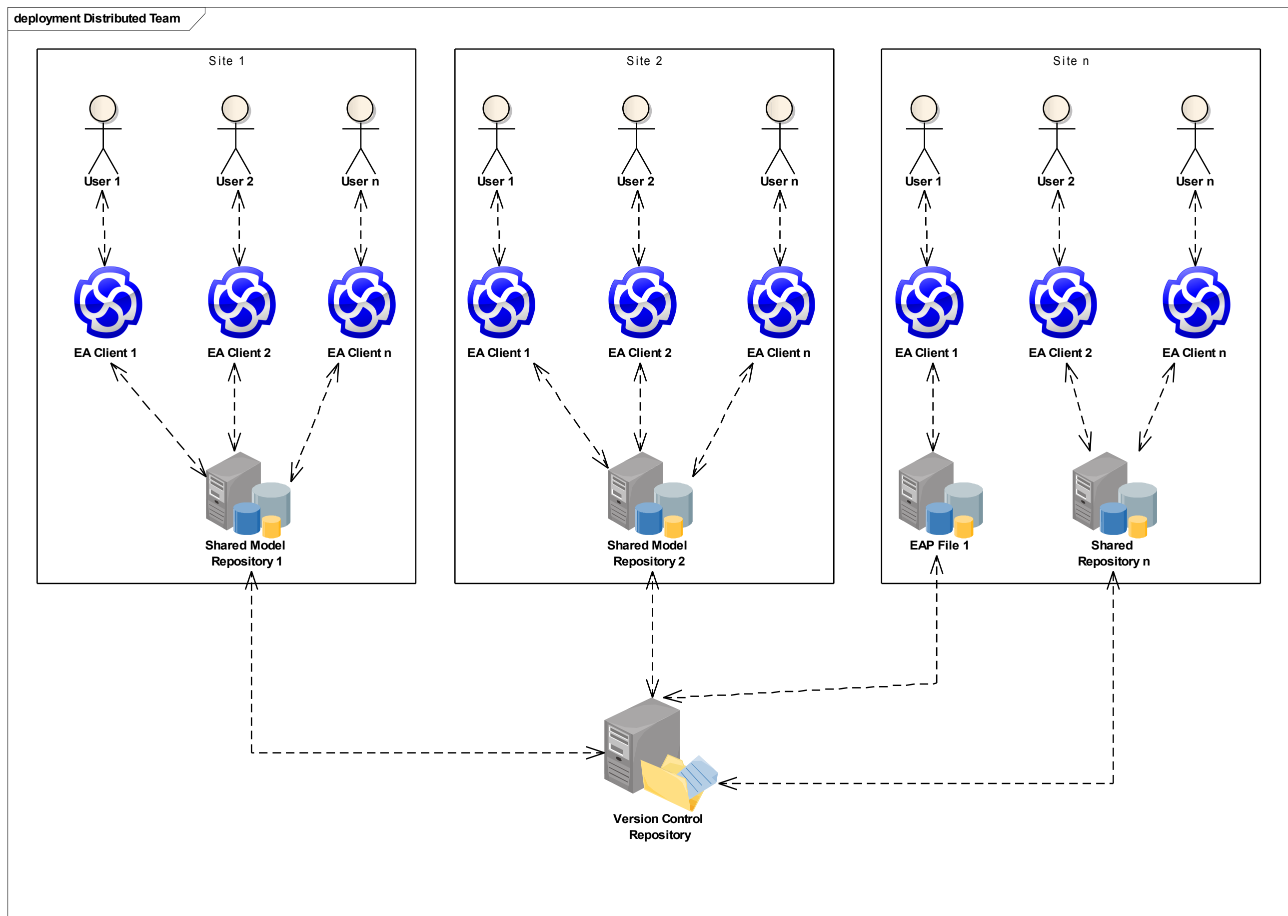

Figure 15: The Version Control Repository can facilitate replication of models across multiple development sites

In multiple site deployments, each shared model repository (described in Scenario 1) assumes a role that is similar to a local repository (described in Scenario 2). The same processes for setting up model repositories in this scenario apply as described in Scenarios 1 and 2 for shared (DBMS) models and local EAP files respectively. The management of cross-Package dependencies also follow the best practices described in Scenario 2.

Consideration must also be given to managing Enterprise Architect's Reference Data so that common project definitions can be shared across each site. Appendix A discusses Reference Data in more detail.

Enterprise Architect provides a convenient mechanism for transferring entire model repositories between sites (including Reference Data) via the 'Project Transfer' feature. For example, you can transfer the model created in the DBMS at site 1, to an EAP file and distribute that file to other sites. These sites can then use the Project Transfer feature to populate an empty DBMS repository by transferring from the EAP file. Enterprise Architect's User Guide describes how to use the Project Transfer feature in detail.

Appendix A: Enterprise Architect meta-data that is not stored in the Version Control Repository.

In this paper, we have considered version management of model Packages and their associated data. It should be noted that Enterprise Architect projects (whether EAP file or DBMS repository) contain additional meta-data, known as Reference Data. This data may used, but not directly defined, within a Package. Examples of Enterprise Architect's Reference Data include: Templates for generating code and Rich-Text Format (RTF) documentation, element Status types and Stereotype definitions.

When such meta-data is used to define the properties of a Package or its elements, that data is exported and imported along with the XMI. Therefore the data is preserved when updating Packages via version control.

It can be useful, however, to make the Reference Data available in the version control repository. Doing so allows all associated Enterprise Architect model repositories to leverage the same definitions and templates. You can use Enterprise Architect's built-in tools to export Reference Data. You can then add the resulting file to your version control repository. Other users may subsequently retrieve the Reference Data and import it into their projects. The export and import processes are defined in Enterprise Architect's User Guide.

Appendix B: Built-in Collaboration and Change Management Tools

We have focused primarily on use of a third-party version control system to replicate model information for distributed teams and to manage model revisions. There are numerous complementary tools provided by Enterprise Architect that facilitate large-scale, team modeling and may be used without a third-party system. We briefly describe some of these tools below:

Auditing

This capability is particularly useful when multiple team members share the same model repository, as it helps to answer: *Who* changed *what* part of the model? *When* was the change made? What was the *previous state*?

Audit information is stored directly in Enterprise Architect's model repository, not the version control repository. The Auditing capability provides a continuous log of changes rather than a point-in-time snapshot. Audit logs can be exported to file and instantaneous changes from logs can be compared directly inside Enterprise Architect. The Enterprise Architect User Guide provides more information on Auditing.

Baseline Compare and Merge

Enterprise Architect can store Package versions directly in the model repository as Baselines. This allows you to compare a Package with a prior state and roll back unwanted changes. Because the Baseline's format is XMI, you can also compare a Package to any XMI file that has previously been exported from that Package. This allows you to selectively merge changes made by colleagues who are working on local copies of the model, without using a version control system. The Enterprise Architect User Guide provides more information on Baselines, Compare and Merge.

Controlled Packages

Packages in Enterprise Architect can be marked as 'Controlled' without being connected to a separate version control system. A 'Controlled Package' is recognized by Enterprise Architect as having a corresponding XMI file with which it may be synchronized. While Controlled Packages have only limited file management commands compared to version controlled Packages, they can be loaded, saved and configured more conveniently than manually performing XMI import/export operations on each Package.

Role-based (User) Security

Enterprise Architect's User Security provides a mechanism for editors to log-in to the model, which serves two important functions. Firstly, it allows organizations to restrict which editing features are available to users. Secondly, it allows Packages and Elements to be locked per-user or per-group. When version control is used in a model only the first function of User Security is applicable, that is restricting availability of editing features. When version control is not used in a shared model however, User Security plays an important role in facilitating collaborative modeling. By applying security locks, teams members avoid overwriting each others' changes and avoid inadvertent model changes by users who are not designated as model authors.

Appendix C: Applying Version Control to Packages

In this appendix we provide suggested approaches for applying version control to Packages, based on what model structure you may already have in place and your deployment scenario. We refer to some of Enterprise Architect's version control features that are more fully explained in the User Guide on configuring a version controlled Package.

Processes for retroactively applying version control to existing Packages

How to version control every Package in the model:

You may wish to add each Package to version control in order to maximize the potential for parallel editing. If that is the case, simply use Enterprise Architect's 'Add Branch to Version Control' command at the model's Root node. In this context, the word 'branch' refers to the model sub-tree beginning at the Package you have selected – so if you apply 'Add Branch to Version Control' at the Root node, the 'branch' in effect, is the entire model.

The result is that version control will be recursively applied to all Packages and their sub-Packages. The corresponding XMI files are automatically named based on the Package's GUID, which survives any subsequent renaming of Packages. XMI files for parent Package's will only contain 'stub' information for child Packages, which reduces individual file size.

Note: The 'Add Branch to Version Control' command will prompt you to "Export as Model Branch" and it is recommended that you take this option. A Model Branch file (*.EAB) gives you a convenient reference to the model sub-tree you are exporting. It is a small file that can be named in human readable terms (as opposed to a GUID). Later, if you or another team member needs to populate a model repository from scratch, you can easily do so via the 'Import a Model Branch' command.

How to selectively apply version control:

Alternatively, if you have a distributed team, you may wish to reduce cross-Package dependencies between version controlled Packages. This requires that you do not independently version control lower level Packages. Instead, lower level Packages get included with the parent Package's XMI file in the version control repository. The process is:

1. Determine which top-level Packages represent sufficiently self-contained model portions.
2. For each Package:
   i. Right-click and choose Configure, or use the keyboard short-cut "Ctrl+Alt+P" (useful when repeating for several Packages).
   ii. Choose the appropriate version control configuration, adjust the default XMI file name if needed and leave the remaining options as default.

Process for applying version control to a new, empty model as it is being constructed.

1. Create a skeletal Package structure for the model.
2. Add Packages to version control using one of the following approaches:
   a) Use the command "Add Branch to Version Control" to apply version control to all Packages or;
   b) Apply version control to individual Packages:
      i. Right-click and choose Configure, or use the keyboard short-cut "Ctrl+Alt+P".
      ii. Choose the appropriate version control configuration, adjust the default XMI file name if needed and leave the remaining options as default.
3. When a new Package is added to the model, the "Create New Package" dialog will present you with an option to determine whether it will be added to version control.

If you are working in a distributed team environment, once the model has been set up:

1. Check-in all Packages and put aside as a "master" model.
2. Distribute copies of this "master" to the team members.