Comparing and Merging Models with DevOps Model RealTime

Author: Mattias Mohlin
Senior Software Architect
HCL

This document describes how to compare and merge models developed with DevOps Model RealTime. It explains the Compare/Merge user interface and suggests suitable workflows to use when resolving various kinds of conflicts that may appear during a model merge.

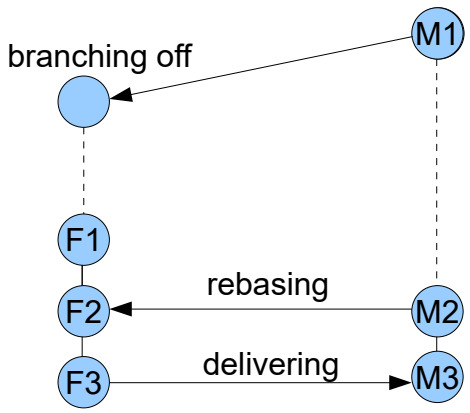
The document was last updated for Model RealTime 10.2. All screen shots were captured on the Windows platform.

Introduction

Most software projects have a need for parallel development, where features are developed in parallel by different development teams. Parallel development is made possible by using a configuration management (CM) system, for example ClearCase, Rational Team Concert or Git, which allows the work of different teams to be isolated from each other. Different CM systems use different terminologies, but in this document we will use the term **stream** to denote the area in the CM system where one particular feature can be developed in isolation from other features. Exactly what constitutes a feature varies between different organizations and projects, and here we will loosely define a feature as a group of changes performed by a certain team over a limited period of time.

It is common to have a main stream, or release stream, from which the different feature streams are branched off. The main stream is where the work of the different feature teams is put together. Usually the complete software is built, tested and delivered from the main stream. This means that contrary to a feature stream which is only important to the team that works on that feature, the main stream is important to everyone in the project. Therefore it is crucial to keep the main stream in a good and healthy condition at all times. Ideally all code on the main stream should always build successfully, all integration tests should pass, and the product that is built from it should always move in a direction where it becomes continuously better and more feature-rich as time goes by.

The picture below illustrates the main stream (to the right) and a feature stream (to the left) for a particular model file:



At version M1 the feature stream is created by branching from the main stream. Version M1 is called the **base version** or **common ancestor version** for the feature stream. The feature is then developed in parallel with all other development activities that take place on other streams. At version F1 the feature has reached a state where it is ready to be introduced on the main stream for integration with other features. Before the development team can deliver the changes from the feature stream to the main stream they have to incorporate all changes that have arrived on the main stream since version M1. This is a merge activity which is usually called **rebasing** since it changes the base version for the feature stream. The result of rebasing is version F2 on the feature stream and the common ancestor version then becomes M2.
Note that rebasing can be done not only when the feature is ready to be delivered. Especially if it takes long time to complete the feature it is a good idea to rebase now and then to avoid

that the contents of the feature stream deviates too much from the contents on the main stream. The more different the feature stream becomes from the main stream, the more difficult will the rebasing be with a higher number of merge conflicts.

Once the rebasing is completed the development team has to ensure that the feature still builds correctly and that all tests pass. Even if all merge conflicts were successfully resolved, there is no guarantee that the resulting model will be fully correct. This can only be verified by building and testing it. Once this has been done, and all found problems have been resolved on the feature stream, the team may try to deliver the changes to the main stream. However, new changes may have arrived on the main stream from other development teams since the point in time when the rebasing was done. If new changes are present on the main stream, another rebasing has to be done to merge these into the feature stream.

Eventually, the feature will be ready to be delivered and there are no new changes on the main stream. This is version F3 in the picture above. Now the feature may be delivered to the main stream (version M3). Note that although this is also a merge activity, it will not lead to any merge conflicts since all changes on the main stream already exist on the feature stream. Such a merge is called a **trivial merge**. All merges to the main stream should be trivial because as soon as new changes are introduced or there are merge conflicts to resolve during a merge, complexity rises and there is a higher risk that the model that results from the merge will fail to build or to pass all test cases. If this happens on a feature stream it is acceptable because it will only affect the team that works on that feature. But if it happens on the main stream it will affect all developers which is much worse. A main stream that has build or test failures means that

- it is not advisable to branch off new feature streams since they won't originate from a stable version
- rebasing from the main stream to feature streams will introduce those errors also on the feature streams
- it is not possible to release the product until the problems on the main stream have been solved

The workflow described above is not at all specific for models - all files that are worked on in parallel are handled in the same way. For textual files Model RealTime includes a tool (part of Eclipse) for comparing and merging them. Compared to this text-based compare/merge tool, the following is worth noticing for the model-based Compare/Merge in Model RealTime:

- Changes and merge conflicts that are reported from models are sometimes harder to understand than those you get when comparing or merging text files. The reason is that models often work on a higher abstraction level than text files. Elements in Model RealTime are described by a meta model (UML2) which uses a rather technical and abstract terminology, that in some cases are different from the terminology used when creating RT models. Descriptions of changes and conflicts may therefore require some experience to understand.
- Changes and merge conflicts often depend on each other. This is because models are highly interconnected and must comply with many integrity rules in order to be well-formed. One consequence of this is that when you resolve a merge conflict, other merge conflicts may be automatically resolved as well. So a merge of models may initially seem complex with lots of merge conflicts, but once some are resolved, the situation may look much better.

- Contrary to text, a model element can be presented in a number of different ways. For example in a model tree (like the Project Explorer), in a property view, in a code view or in a diagram. Depending on the kind of change different ways of presenting it may be appropriate. The Compare/Merge user interface therefore has significantly more compartments and buttons than its textual counterpart has.
- When merging models Model RealTime will not allow you to edit the result model directly. If a conflict cannot be resolved by the commands that are provided by the Compare/Merge tool, you may have to take a note of the affected element's location and return there after the merge has been completed, in order to manually correct or complete that part of the merged model. Model RealTime provides a specific user interface, the Compare/Merge Tasks view, which helps you keep track of such post-merge activities.

The rest of this document explains the user interface of the Compare/Merge tool and describes how to best use it in order to compare and merge models. For the sake of simplicity we will often refer to the workflow with feature and main streams described above, although the Compare/Merge tool really is independent of which work flow that is used for parallel development.

Scenarios that involve usage of a CM tool are described for both ClearCase and Git. For ClearCase it is assumed that the ClearTeam Explorer plugin is used for performing CM operations from within Model RealTime. For Git it is assumed that the EGit plugin, together with the RSx EGit Integration, is used instead. Both ClearTeam Explorer and EGit integrates with the Eclipse Team API which gives their user interfaces certain similarities. It should therefore be relatively easy to understand these chapters even if you are using a different CM system, with another plugin that integrates with the Eclipse Team API.

Comparing Models

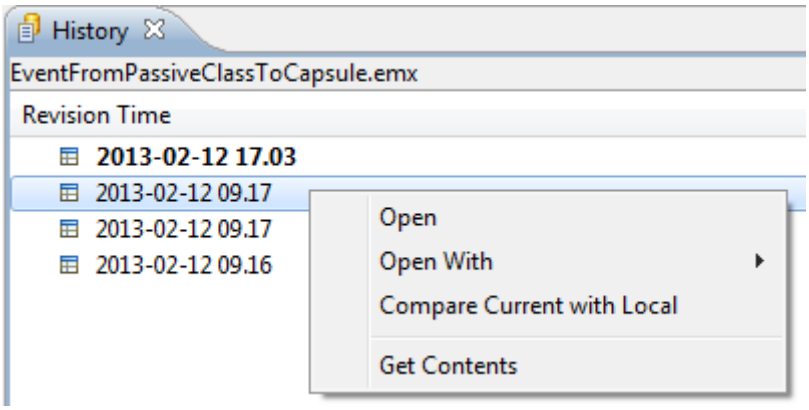
Let's start with looking at how to compare two models (or more accurately, two versions of the same model). This scenario is simpler than the merge scenario where in addition to these two models we also have a third common ancestor model for the base version. It is therefore good to start by learning how to compare two versions of a model.

The two models that are compared are called the **contributor** models. One of them will be shown to the left in the Compare user interface, and the other to the right, and the models are therefore often referred to as the **left contributor** and the **right contributor** respectively. The Compare tool will tell you what changes that are needed in the right contributor to make it become equal with the left contributor. This means that if one of the compared models can be said to be a newer, or more recent, version of the other compared model, then the left contributor should be the newer version of the model, and the right contributor should be the older version of the model. This ensures that you can read the changes that are presented by the Compare tool and understand what has happened with the model from its older version to its newer version.

You can trigger a compare of two versions of a model in many different ways in Model RealTime. Let's look at some of these possibilities before we describe the Compare user interface.

Compare from Local History

Model RealTime has a feature to save the history of workspace files that are modified. This feature can be set-up in the preferences at *General - Workspace - Local History*. With this feature enabled Model RealTime will save a copy of each file that gets modified, and thereby create a history of all changes made to that file. You can look at this history by right-clicking in the Project Explorer on a model that is stored in its own file, and perform the command *Team - Show Local History*.



The version shown in boldface is the current version of the model. If you right-click on a previous version and perform *Get Contents* you will create a new version of the model which is a copy of the selected version. This is useful in case you want to go back to a previous version of a model, and you have not stored that version in the CM system.

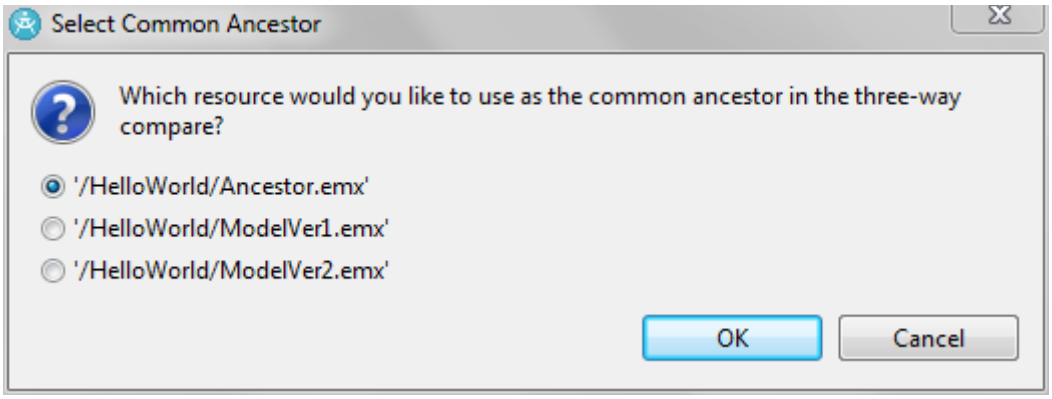
If you do *Compare Current with Local* Model RealTime will launch the Compare tool to show the differences between the current version and the selected version of the model. You can also select two arbitrary versions in the list and do *Compare With Each Other* to compare those particular versions of the model.

When you compare models using the local history feature, the left contributor will be the newest of the two models.

Compare from File System

Comparing files in the local history is a special case of the more general workflow when model files from your local file system are compared. You can select any two elements in the Project Explorer that are stored in their own files and invoke *Compare With - Each Other* in the context menu. The Compare tool will be launched to show the differences between the selected models. To give a useful result, the selected model files must be versions of the same model.

You can also invoke the *Compare With - Each Other* command with three model file elements selected. In this case one of the model files should be the **common ancestor** of the other files. The common ancestor model is the model from which both the two compared model versions originate. Read more about this in Merging Models. A dialog appears to let you choose which of the selected model files that is the common ancestor model:



A compare that involves a common ancestor version of the model is called a **three-way compare**. If there is no common ancestor specified the compare is two-way. The Compare tool can in general do a better job with identifying and presenting the changes that have been made if a common ancestor model is available. This is usually the case when doing compare on files stored in a CM system.

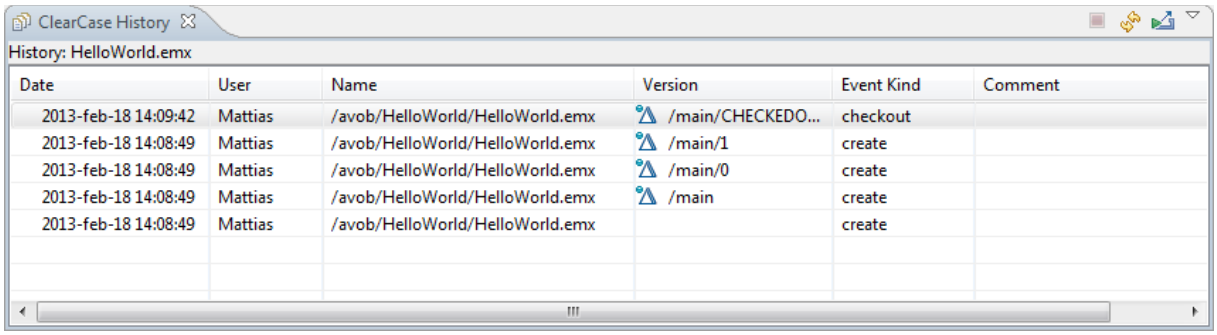
You can also compare files in the file system from the command-line using the Command Line Tool for Compare/Merge.

Compare from CM System

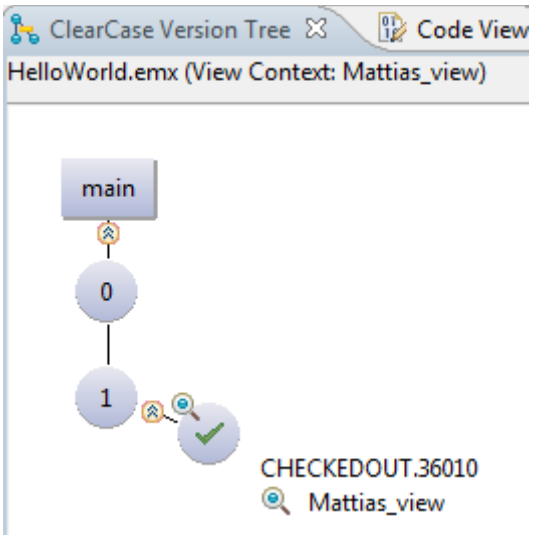
For files that are controlled by a CM system, you can in addition to the local history (which is stored in the workspace) also look at the history of the file that is stored in the CM system. From there you can launch the Compare tool on different versions in much the same way as you can from the local history view. However, in a CM system you can have more than one parallel stream of development, which means that the history of a file is typically better shown as a tree rather than a list. Different CM tools provide different history views.

ClearCase

Right-click in the Project Explorer on a model that is stored in its own file, and perform the command *Team - Show History*. A view will appear that shows the versions of the file in ClearCase:



This view shows the versions of the file as a flat list. To instead view the file's version tree, right click on a file version in the list and do *Show Version Tree*. Here is an example of a very simple version tree for a model file "HelloWorld.emx" which is controlled by a ClearCase dynamic view:



Here we can see that the main stream has two versions of this file (version 0 and 1), and that the file is currently checked out to allow a third version to be created. To launch the Compare tool on versions of the model file that is shown in the version tree you can right-click on one version and perform *Compare With Predecessor*. The left contributor will then be the selected version and the right contributor will be the predecessor version. You can also perform *Compare With Another Version*, and then click on the version you want to compare against. The version you invoke the command on will be the right contributor model and the version you click the next time will be the left contributor model. It is therefore important to select the versions to compare in the correct order, so that you start with the oldest version first. For example, if we would like to see what has happened from version 0 to version 1 in the picture above, we should first select version 0 (right contributor) and then version 1 (left contributor).

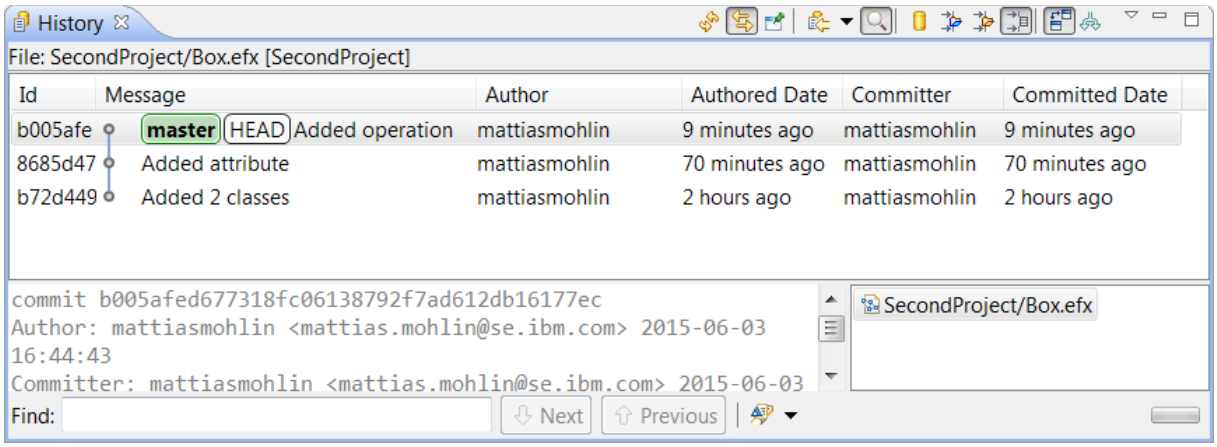
You can also launch the Compare tool from the command line. For example, the equivalent of the above mentioned compare operation would be

```
cleartool diff -graphical HelloWorld.emx@@\main\0 HelloWorld.emx@@\main\1
```

Note that also here the first specified version will be the right contributor, and the second specified version will be the left contributor. See Compare/Merge Server for more information about how command-line invocation of Compare/Merge works.

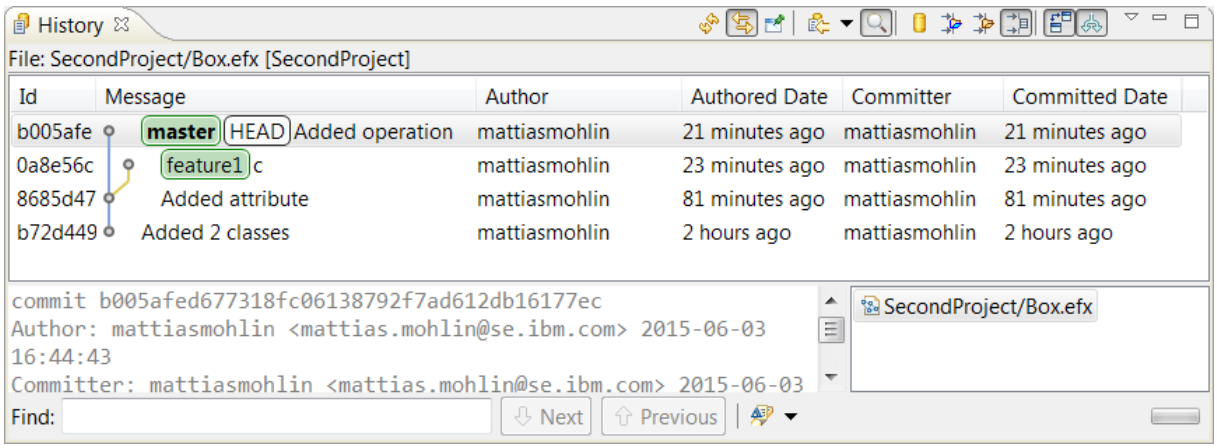
Git

Right-click in the Project Explorer on a model that is stored in its own file, and perform the command *Team - Show in History*. A view will appear that shows the versions of the file in Git:



The Git history view actually shows the history of commits, and by default when you invoke it on a particular model file the commit history is filtered to only show the commits that contain changes in that file. You can use the buttons in the view toolbar to widen the scope so that you can look at the commit history for the parent folder, the containing project or even the entire Git repository.

To view the full version tree, rather than a flat list of commits, click on the toolbar button *Show All Branches and Tags*. The History view will then not only show the list of parent commits for the checked out commit, but also commits on other branches:



You can compare any commit with the version of the model you currently have in the workspace. Right-click on the commit and perform *Compare with Workspace*. The right contributor will be the selected commit, and the left contributor will be the workspace version.

More variants of this command are available in the *Compare With* context menu for an element in the Project Explorer that is stored in its own file. The commands in that menu can be used for comparing the model of the selected file with the version of a particular commit, branch, tag or reference. You can also compare it with the staged version that is currently in the Git index and with the HEAD revision. *Compare Index with HEAD* compares the version of the file that is in the index with the HEAD revision.

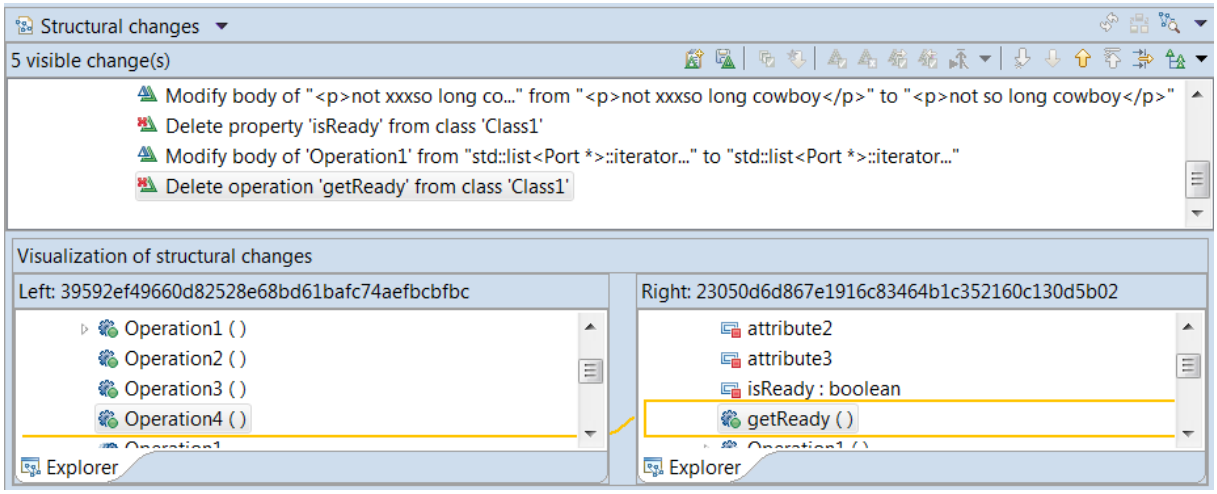
If you select two commits in the History view you can compare them by performing *Compare with Each Other* in the context menu. The right contributor will be the commit that has the oldest authored date, and the left contributor the one with the newest date.

A compare session can also be launched from the Git Staging view which shows files that are currently unstaged (locally modified) and staged (added to the index). You can simply double-click on a file shown in this view to launch a compare session which shows the changes in the file compared to the HEAD revision.

It is possible to launch the Compare tool from the Git command line. You need to configure the git difftool to invoke the Command Line Tool for Compare/Merge.

Compare User Interface

The Compare user interface consists of three compartments:

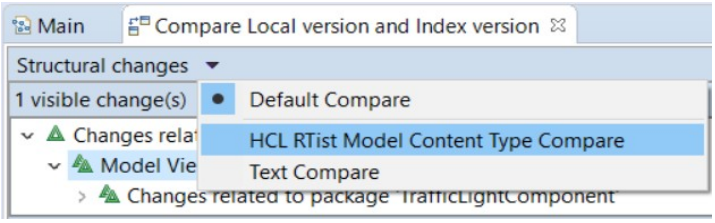


The top compartment shows the list of changes, and the bottom two compartments show the left and right contributor models respectively. For each change that you select in the change list, it will be highlighted in both the left and right contributor models, so you can see where in the model the change was made.

Hint: You can double-click the header of a compartment to maximize that compartment and hide all the others. Double-click the header again to make all compartments visible.

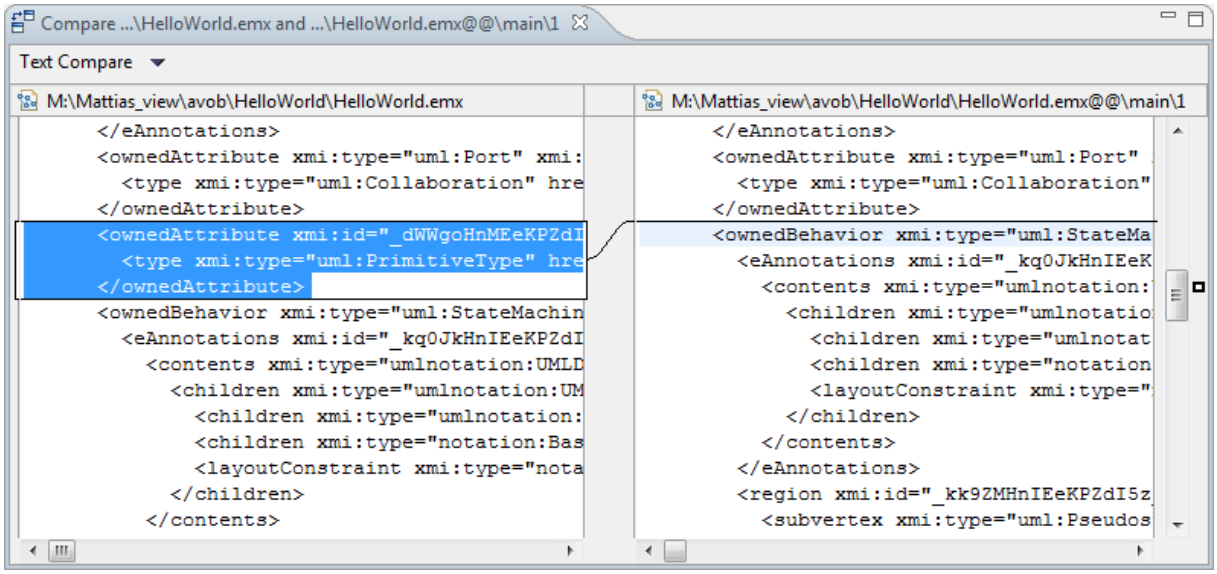
As you can see there are several buttons and menus available in the Compare user interface. Many of these apply only for merge sessions, and are therefore disabled for a compare session. Let's go through the buttons and menus that are enabled for compare sessions:

Switch Compare Viewer



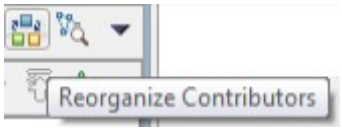
The Compare tool supports comparing many kinds of files, not just UML model files created in Model RealTime. For each kind of supported file one or many **content types** may apply. Each content type corresponds to a particular **compare viewer** which controls the overall appearance of the Compare user interface.
The Compare tool uses the file extension of the compared files to find a default content type to use for the compare session. For example, for model files (.emx, .efx etc.) the default content type is called "Model RealTime Model Content Type". When this content type is used, the Compare user interface will consist of the three compartments that were mentioned previously.

You can use the menu that appears immediately to the right of the text "Structural changes" to switch to using a different content type and compare viewer. As you can see in the picture above, the content type "Text" is also available for model files. If you choose this content type the Compare tool will treat the compared files as plain text files and the Compare user interface will change to just show two compartments, where the models are shown as text files:



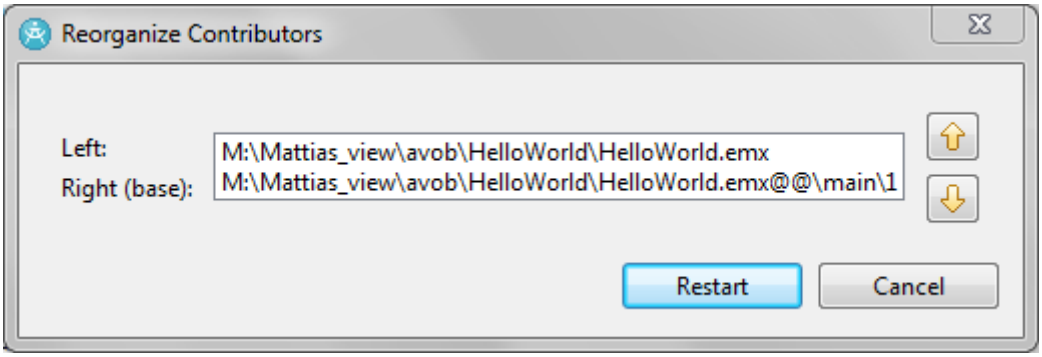
Switching to a non-default content type, such as Text, can sometimes help you better understand what a particular change relates to. However, in most cases the default content type will be the most appropriate to use.

Reorganize Contributors



Before starting to look at any of the changes in the change list, it's a good idea to first check so that the correct model versions were chosen as the left and right contributor. As said previously Compare/Merge reports changes as "what happened to the left contributor compared to the right contributor" or, put differently, "what changes are required to be made in the right contributor to make it equal with the left contributor". If you by mistake choose the contributors in the wrong order when doing a compare, you will see Add changes when Delete

changes were expected, and vice versa. If this happens you can easily swap the order of the contributor models by using the Reorganize Contributors button. When pressing the button a dialog appears that lets you swap the order of the contributor models:



If the compare is three-way an ancestor contributor will also be shown here.

Close the dialog by pressing the Restart button to relaunch the compare session with re-ordered contributors.

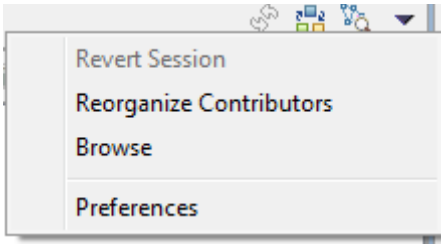
Browse Button

The Browse button is a toggle button which, if pressed, switches the Compare user interface to a so called browse mode. In this mode only two compartments (three in case of three-way compare) are visible, one for each contributor model. In the browse mode you can explore the contributor models in more detail than what is otherwise possible. Each compartment will show three tabs that let you explore the model in different ways:

- **Explorer**
  Explore the model using a tree viewer similar to the Project Explorer.
- **Properties**
  Explore all details of one particular element in the model using a property page similar to the Advanced tab in the Properties view. To use the Properties tab first select an element in the Explorer tab, then right-click and choose the *Show in Properties* command.
- **Diagram**
  Explore a diagram of the model. To use the Diagram tab first select a diagram in the Explorer tab, then right-click and choose the *Show in Diagram* command.

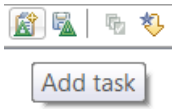
Note that you can also explore the contributor models using the regular Properties view and Code view. See Exploring the Contributor Models to learn more about this.

Preferences Menu



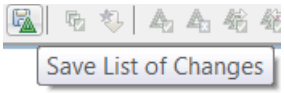
This menu is located in the top-right corner of the Compare user interface. It contains a *Preferences* command that opens the preference page for Compare/Merge. The menu also repeats some of the commands from the toolbar.

Add Task

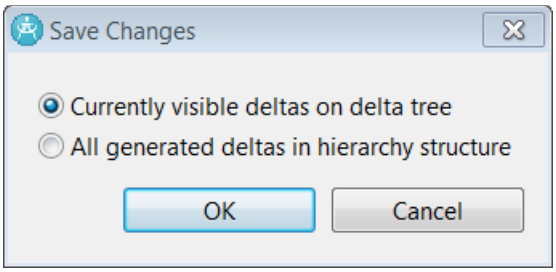


This button creates a Compare/Merge task for the change that is selected in the change list. You can for example use this as a means of reviewing changes in a model. See Compare/Merge Tasks for more information.

Save List of Changes

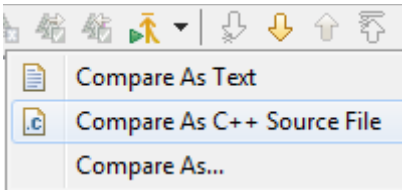


This button allows you to save the list of changes to a textual log file. You will be asked if you want to save only the changes that are currently visible (i.e. not the ones that are hidden because you have applied a filter), or if you want to save all the changes.



The dialog uses the term **delta**, which is just a fancy name for a change. See Delta Tree Configuration for more information about how the change list can be filtered.

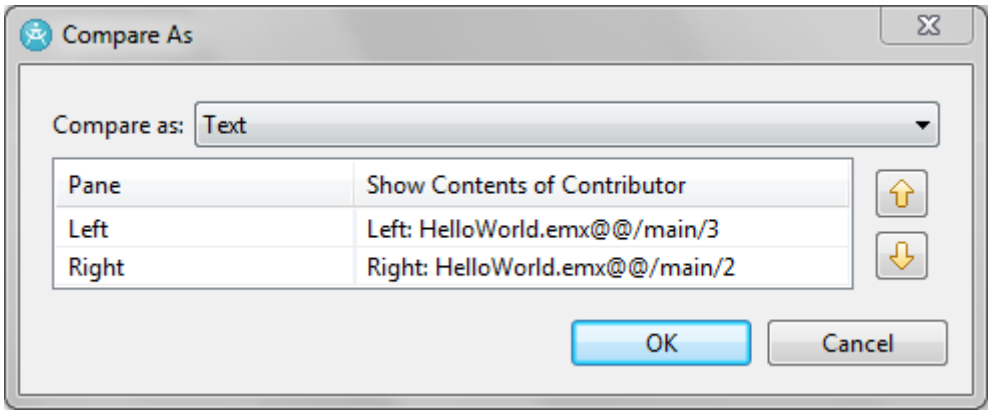
Sub Compare Button



This button becomes enabled when you select a change in the change list that relates to a changed code snippet or documentation comment. Such changes are best explored using a textual sub compare session. See Sub Compare for more information about what this is.

For a code snippet that contains C++ code the default sub merge command is *Compare As C++ Source File* which means that the compare tool provided by CDT will be used for comparing the code snippets. However, you can also do the sub compare by choosing *Compare As Text* in the button menu. In this case the base text compare tool provided by Eclipse will be used.

If you choose *Compare As...* in the Sub Compare button menu a dialog appears that lets you choose both which tool to use for the sub compare, and also in which order the contributors should be set for the sub compare session:

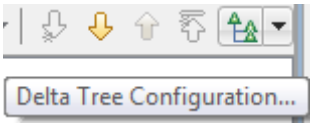


Usually there is no reason to swap the contributors with this dialog since the default order is the same as is used for the corresponding left and right contributor models in the enclosing model compare session.

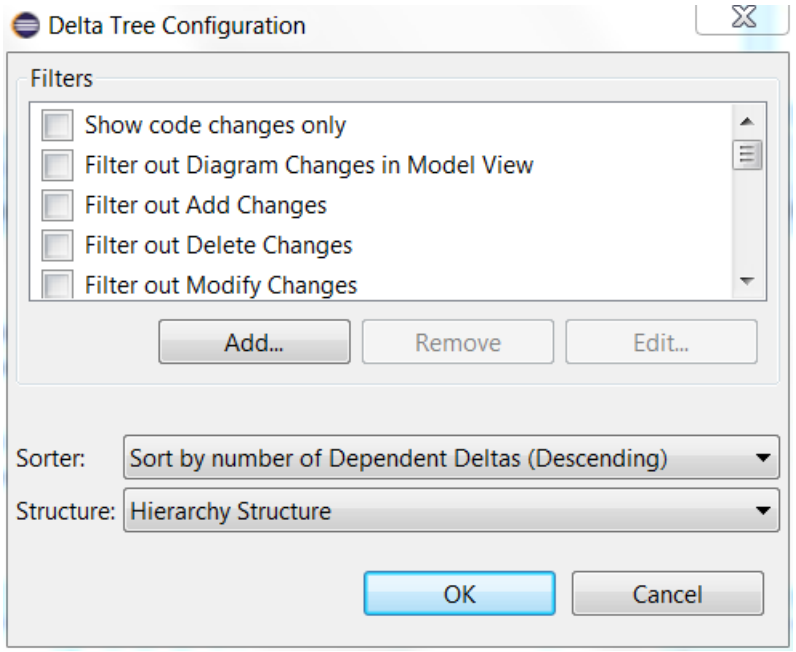
Next/Previous Buttons

You can use these buttons to go through the changes in the change list, as an alternative to clicking on the changes one by one.

Delta Tree Configuration



Sometimes the number of changes in the change list can be so large that it becomes impractical to show them all at the same time. In this case you can use the *Delta Tree Configuration* button to apply a filter so that some of the changes become hidden. If you press the button the following dialog appears:



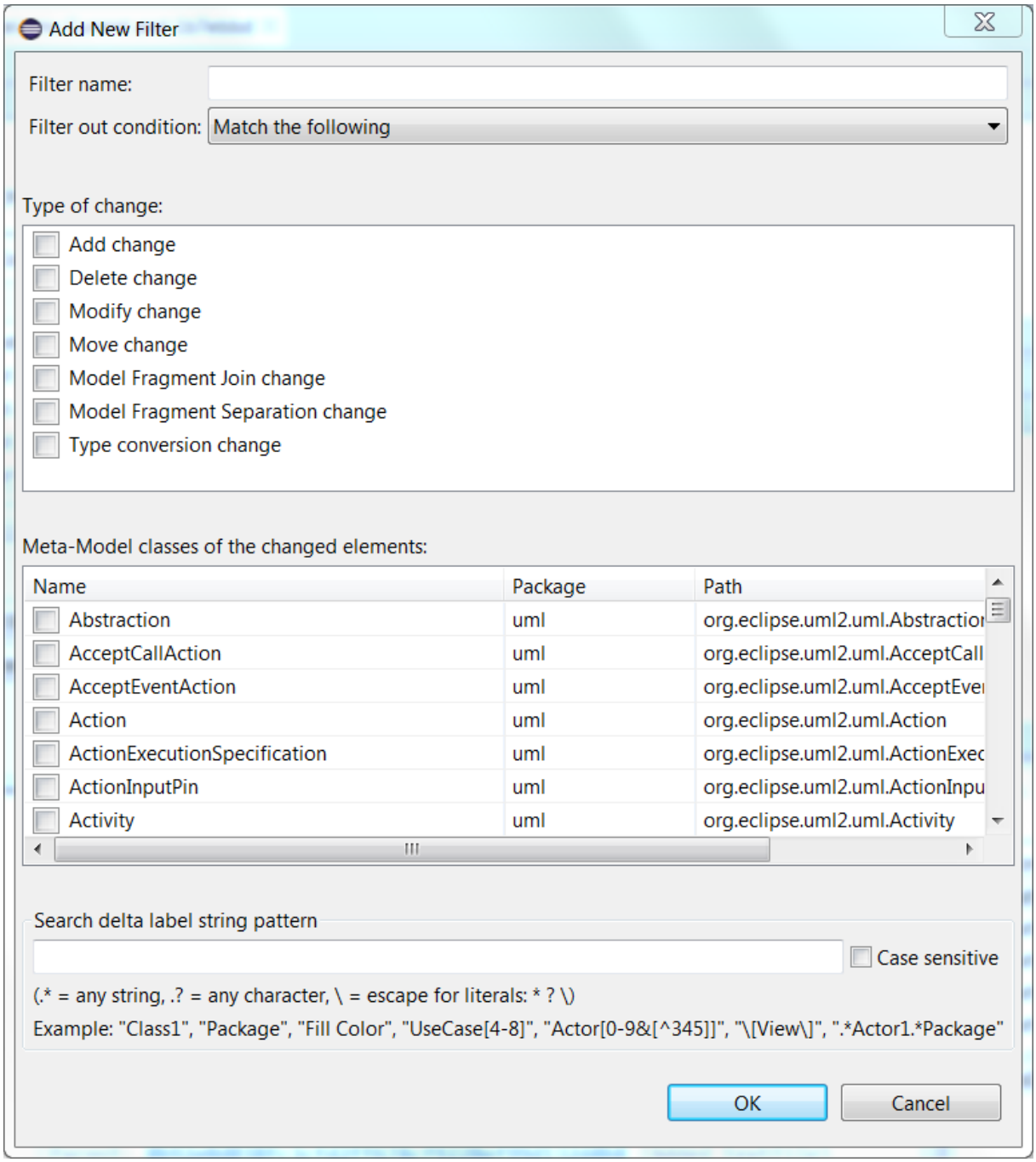
The bottom drop down menus allow you to configure how to sort and structure the changes that are shown in the change list. The default sorting is to sort according to the number of dependent changes, so that changes that have many dependent changes appear before changes that have few dependent changes. See Dependent Changes for more information about what a dependent change is. You can also choose to sort by model element name, by the change description ("Delta Label") or not to sort at all. The default structuring of changes is "Hierarchy Structure" which means that changes are grouped into a hierarchy corresponding to the hierarchy of the changed model elements. There are several other alternatives for how to structure the change list, including "Flat Structure" which means that changes are not grouped at all but appear in a single flat list.

The Filters list shows all the different kinds of changes that may appear in the change list. See Types of Changes for more information about what the different kinds of changes mean. Note also that some filters simply work by filtering out all changes that are related to a certain kind of element, regardless of the type of changes:

- The "Diagram Changes" filter means that all changes that are related to diagrams will be filtered out (everywhere, or only under the “Model View” node)
- The "Documentation Changes" filter means that all changes that are related to documentation comments will be filtered out.
- The "Stereotype Application Changes" filter means that all changes that are related to applied stereotypes will be filtered out.

It can also be useful to filter out all changes that are not related to code changes, by choosing “Show code changes only”.

You can create your own custom filter by pressing the Add button:

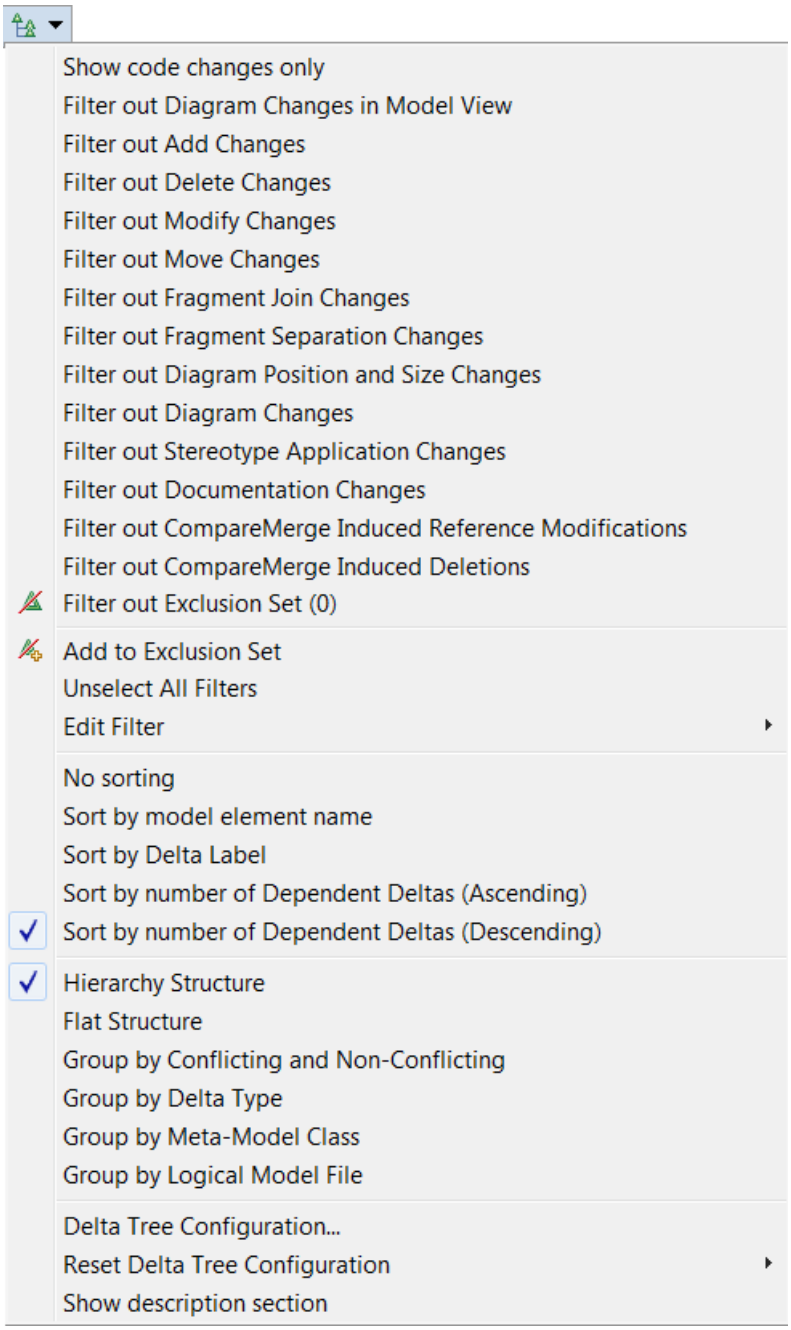


This dialog lets you create a filter that

- filters out changes of one or many types (mark the checkboxes in the upper list for which types of changes to filter out)
- filters out changes that relate to model elements of certain kinds (mark the checkboxes in the lower list for which kinds of model elements to filter out)
- filters out changes that have a certain kind of description (enter a regular expression in the bottom text box which will be matched against the description of changes, and hide those changes that match)

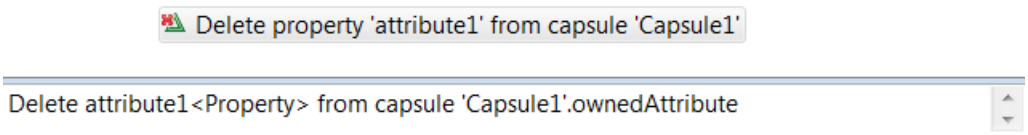
For example, if you want to filter out all changes that relate to attributes that start with the prefix "He" you may create a filter with the meta-class "Property" selected in the lower list and with the following description pattern: `He*`

The Delta Tree Configuration button also has a button menu that lets you turn on or off filters quickly:



The commands in this button menu let you do the same kind of filtering, sorting and structuring of the change list as the Delta Tree Configuration dialog described above. In addition it contains a few other commands that affect what is shown in the change list:

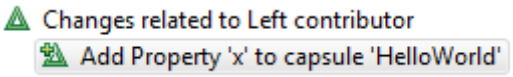
- *Add to Exclusion Set*
  This command can be used to add a selected change to a predefined filter called "Exclusion Set (N)" where N is the number of changes that have been added to this filter. This is hence a way to filter out individual changes from the change list.
- *Show description section*
  This command toggles visibility of a text box at the bottom of the change list which contains a more detailed description of the selected change. Here is an example:



  The detailed description text can be quite technical, and by default this text box is not shown to save user interface space.

Change List Compartment

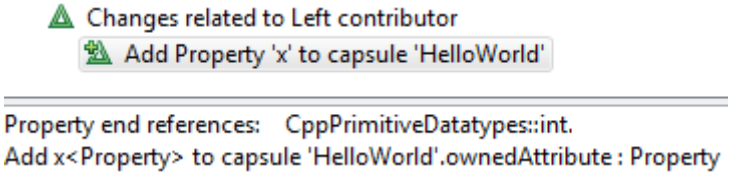
The change list shows all (or a subset of, if a delta tree filter has been applied) the changes between the left and right contributor models. The changes are always presented as related to the left contributor model. For example, if the left contributor contains an attribute that the right contributor does not contain, the change list will contain an Add change for that attribute:



You can right-click on a change in the change list and perform the *Navigate – Show in Project Explorer* command in order to try to locate the changed model element in the model that is loaded in the Model RealTime workspace. Doing so can help you better understand the context of a particular change, but the navigation will of course only work if the changed element exists in the workspace model. Depending on the kind of changed element there may also be other useful navigation commands in the *Navigate* context menu. For example, if the changed element is shown on a diagram you can navigate to that diagram.

The description text for a change in the change list can sometimes be quite long. If the text is too long to fit in the change list compartment, you can look in the Model RealTime status bar which also shows the description of the selected change.

Another approach Model RealTime uses to avoid very long descriptions of changes is to put some of the information about the selected change in a separate text area. This text area is located just below the change list. For example, for the attribute shown in the picture above the text area will contain text that tells you what type the added attribute has:



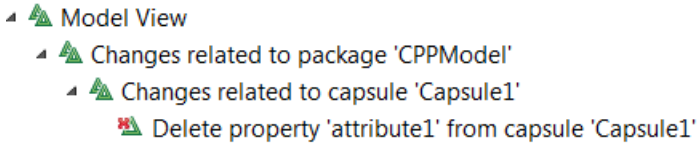
Here we can see that the type of the added attribute is the primitive C++ type 'int'. There may also be a more detailed technical description of the change. For example, above we can see that the attribute "x" was added as one of the owned attributes of the capsule "HelloWorld". Note that this text area is by default hidden (to save user interface space). You can make it visible by means of the command *Show description section* in the Delta Tree Configuration button menu.

The changes shown in the change list compartment are by default grouped into a hierarchy. You may encounter the following groups of changes:

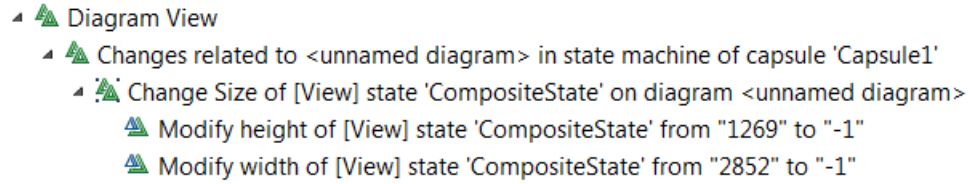
- **Model View**
  Groups all changes to the model that do not fit under any of the other groups. These changes affect the meaning of the model.
- **Diagram View**
  Groups changes related to diagrams. These changes do not affect the meaning of the model, but only how the model is presented in diagrams.
- **Stereotype Application View**
  Groups changes related to applied stereotypes. Usually changes in applied stereotypes do not change the core meaning of the model, but rather affect the additional information which the applied stereotypes constitute. Such information may for example be used by domain-specific tools that operate on the model.
- **Fragment View**
  Groups changes related to model fragment files. These changes do not affect the meaning of the model, but only how the model is stored in files.

Note that grouping is also used within the top-level groups mentioned above. The purpose of such sub-groups is

- to give context to a change. For example, the location of a change in the model is shown by grouping the change under nodes that tell in which class/capsule and package the changed element is located.

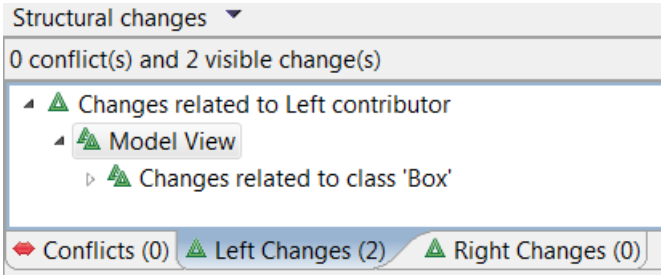
- to group multiple related changes. For example, a simple editing operation such as resizing a symbol on a diagram typically causes two changes (one for the change in width, and another for the change in height), and by grouping them under a single parent “Change Size” change it becomes easier to quickly understand how the model has changed.

If you prefer to not group changes into a hierarchy you can select "Flat Structure" in the Delta Tree Configuration button menu. In this menu you also have choices for sorting the list of changes.

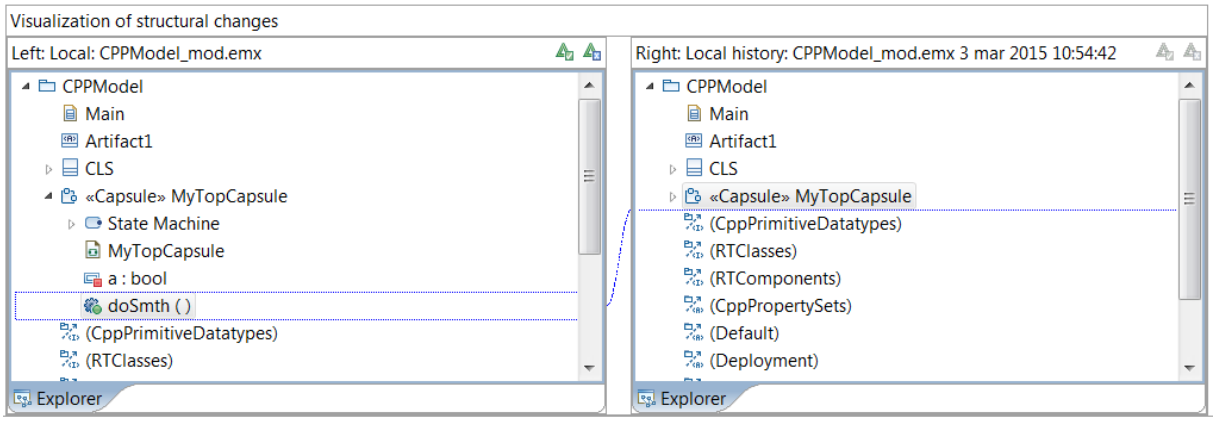
In case of a three-way Compare session the change list compartment will have tabs that show three lists of changes:

- **Left Changes**
  The changes between the left contributor and the common ancestor.
- **Right Changes**
  The changes between the right contributor and the common ancestor.
- **Conflicts**
  The list of those changes in the Left Changes and Right Changes tabs which are conflicting.



Contributor Model Compartments

These two compartments show the left and right contributor models. When you select a change in the change list the changed element is highlighted in both the left and right contributor model in order to visualize the impact of the selected change. If the changed element does not exist in one of the contributors, the nearest container element is highlighted instead. Also, a line is drawn between the contributor models to indicate how they are related by the selected change. For example, if a change describes that an operation "doSmth" was added in the capsule "MyTopCapsule" in the left contributor model, the contributor model compartments may look like this:



The contributor model compartments provide different tabs in order to present the change in different ways. Depending on the type of change that is selected in the change list, different tabs will be available. Sometimes you may have to switch to a different tab to fully understand exactly what has changed. The following tabs are used:

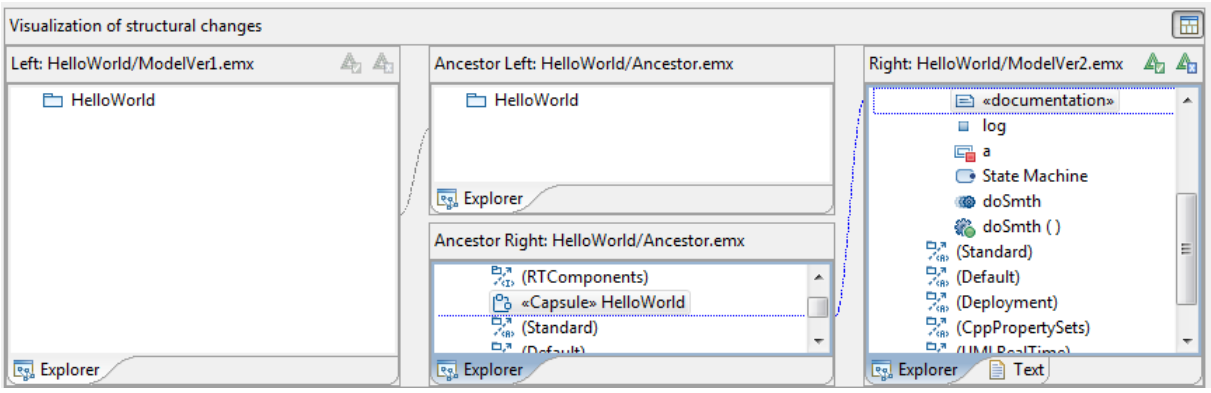
- **Explorer**
  Shows the change in a tree viewer similar to the Project Explorer. This tab is useful for

seeing where in the model the change belongs. This is the default tab for most kinds of changes.

- **Text** Text
  Shows the change in a text viewer. This tab is useful for changes in textual properties such as code snippets or documentation comments. However, a sub compare of a textual property may be even more useful. You can launch a sub compare from the Text tab by right-clicking in the text viewer and selecting the command *Show Text Sub Merge Pane*. See Sub Compare for more information.
- **Properties** Properties
  Shows the change in a property page that is similar to the Advanced tab of the Properties view. This tab is useful for changes in properties that are not textual, or that contain rather short texts (such as the name of an element). For changes in textual properties that contain longer texts the Text tab is more appropriate.
- **Diagram** Diagram
  Shows the change in a diagram. This tab is useful for changes related to diagrams, such as when a symbol has been repositioned or a line has been rerouted. The change is shown in the diagram using a colored ellipse. The colors that are used have the following meaning:
  - Green. Used for Add changes.
  - Grey. Used for Modify changes.
  - Red. Used for Delete changes.

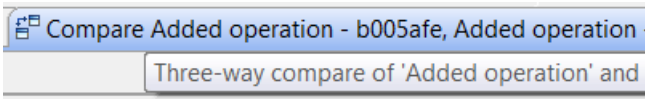
  In case of a three-way compare the Diagram tab has a button *Show Ancestor Diagram* ( ) in the upper right corner which allows the diagrams from the ancestor and the contributor models to be overlaid graphically. This can sometimes help you understand what was changed in the diagram.
- **EMF List** EMF List
  This is an advanced tab that sometimes shows additional low-level information about a change. You usually do not need to use this tab, and it will only appear if the preference *General - Compare/Patch - Modeling Compare/Merge - EMF Compare/Merge - Display EMF tabs in Compare/Merge Editor* has been set.

In the case of a three-way compare (i.e. when a common ancestor model is available) there will be a button called "Show All Contributor Panes" ( ) in the upper right corner of the contributor model compartments. If you press this button you will see the common ancestor model in the middle between the contributor models. You will also see how each change in the contributor models relates to the common ancestor model by means of lines drawn between the compartments. It can for example look like this:



This extended view is possible to show only when doing a three-way compare, because then two sets of changes are computed by the Compare tool:
1. The changes between the left contributor and the common ancestor.
2. The changes between the right contributor and the common ancestor.

So if you are unsure whether you are actually doing a two-way or three-way compare, you can check if the "Show All Contributor Panes" button is visible or not. This information is also shown in the tooltip for the Compare editor:

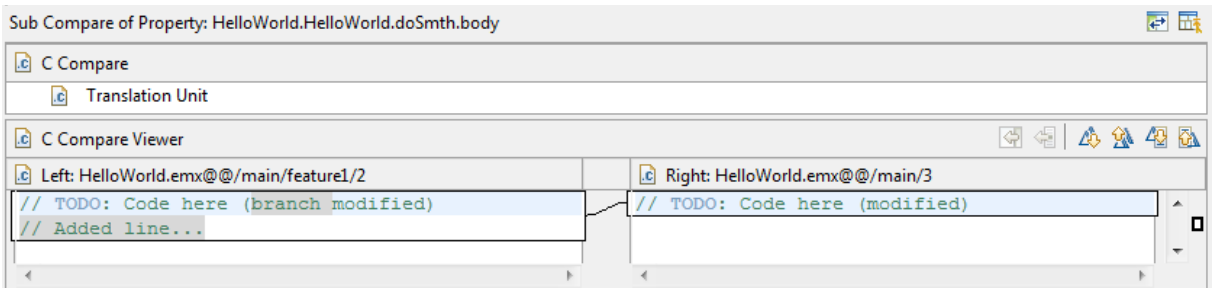


Sub Compare

Some of the changes you will find in the change list may be Modify changes where the modified property contains a piece of text. A typical example is when a code snippet has been modified on a model element. A code snippet can be a fairly large piece of text, and viewing these changes using the Text tab in the contributor model compartments of the Compare user interface is not so easy. To get a better visualization of such textual changes you should instead do a sub compare on the change.

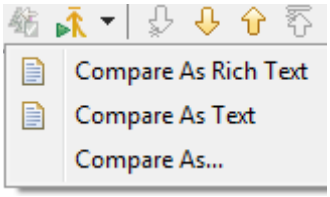
A sub compare is a textual compare session that is "nested" within the model compare session. When doing sub compare a text-based compare tool is launched in order to show the differences that are located in pieces of text, such as code snippets. For C++ code snippets, sub compare by default uses the compare tool that is provided by CDT. Hence you can compare changed C++ code snippets in the same way as you compare two changed C++ files.

Sub compare starts automatically when you select a change that is related to code snippets. The textual compare tool that sub compare uses replaces the two contributor model tabs. It may look like this:

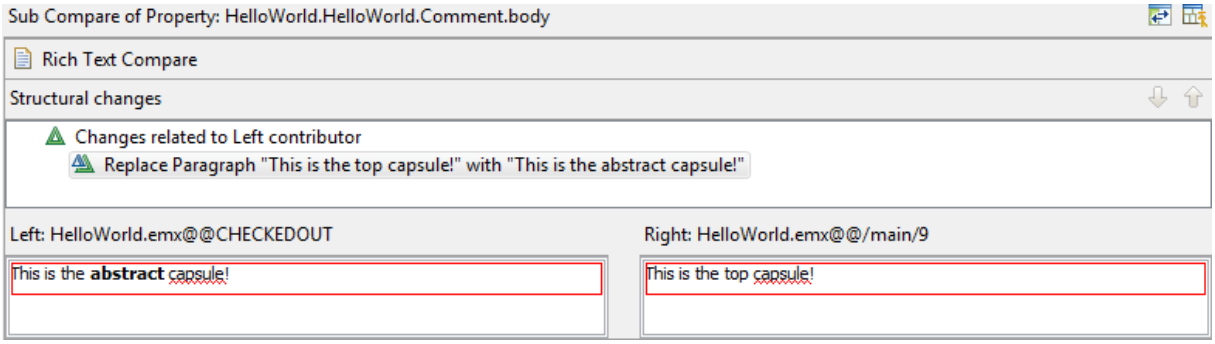


To stop the sub compare session you can press the button *Show Visualization of Structural Changes* ( ). If you later want to go back to sub compare you can press the button *Show Text Sub Merge Pane* ( ). This command is also available in the context menu of the Text tab.

In addition to code snippets it is possible to launch sub compare on changes related to documentation comments. This is especially useful if the documentation comment contains rich text. To launch sub compare on a documentation comment choose *Compare As Rich Text* in the Sub Compare button menu:



The rich text sub compare tool understands the contents of rich text comments better than the plain text compare tool does, and can highlight the changes in the comments in a better way:



Merging Models

Let's now look at the slightly more complex task of merging two models (or more precisely, two versions of the same model) into a third copy of the model. Just like when comparing models we refer to the two models that are merged as the left and right contributor models respectively. The model that is created by the merge operation is called the **merge result model**. In addition to these three models there is also a fourth model involved; the **common ancestor model** from which both the contributor models originate.

The Merge tool starts by comparing the contributor models against the common ancestor model to get the list of changes for each contributor. Then it analyzes these changes to see if any of them are conflicting. A **merge conflict** consists of two changes, one from each contributor, that cannot both be applied to the merge result model at the same time. To complete the merge session you have to resolve all merge conflicts by deciding, for each one of them, whether the change from the left or the right contributor should apply. You can also resolve a conflict by deciding that neither of these changes should be included in the merge result model.

All non-conflicting changes are by default applied automatically by the Merge tool. That is, the Merge tool assumes that you want to have all changes from both the left and the right contributor models in your merge result model. Usually this is what you want but in some cases this is not appropriate, and you may want to skip some of the changes. For example, during the merge session you may realize that the authors of the left and right contributor models may have solved the same problem in two different ways. Their changes may not conflict on a structural level (i.e. they have not modified the same part of the model), but still it does not make sense to have both changes in the merged model. This is an example of a **logical merge conflict**. The Merge tool can only detect **structural merge conflicts**; it cannot help you to detect logical merge conflicts (except in some simple special cases, as mentioned in Structural Merging by Combining Models). There is no guarantee that just because no merge conflicts

are reported, the merge result model will be logically correct. This is something only a human being can confirm, and it is therefore recommended that you review all changes that will be applied to create the merge result model, and also to review what the merge result model will look like after the merge.

A special case of a logical merge conflict is when the standard UML allows something which is not supported in the RT subset of UML. For example, in standard UML a composite state is allowed to have multiple regions, each of which can contain a sub statemachine for the state. However, in Model RealTime only one region with one sub statemachine is supported. If you merge two versions of a model where in both versions a non-composite state has been made composite by adding a sub statemachine for it, then the merge result model will contain both these sub statemachines. Such a model is valid according to UML rules, but invalid according to the more strict UML-RT rules. The reason for problems like this one is that the Model RealTime Merge tool is not aware of all RT specific constraints and hence misses certain logical conflicts. Therefore it is sometimes necessary to edit the merge result model after the merge to make it a correct UML-RT model. By carefully reviewing the merge result model prior to completing the merge session you avoid surprises and can detect places in the model where post-merge editing may be required. You may use Compare/Merge Tasks to help remember changes that are required after completing the merge session.

A merge session that does not contain any merge conflicts (i.e. a trivial merge), and where you are confident that there also will not be any logical merge conflicts, can be performed without using the Merge user interface. We call this an **automatic merge** since the Merge tool then can complete the merge session automatically without any user interaction. One example when it is safe to do an automatic merge is when delivering from a feature stream to the main stream, where you just previously have rebased so that all changes from the main stream already were merged into the feature stream.

You can trigger a merge in many different ways in Model RealTime. Let's look at some of these possibilities before we describe the Merge user interface.

Merge from File System

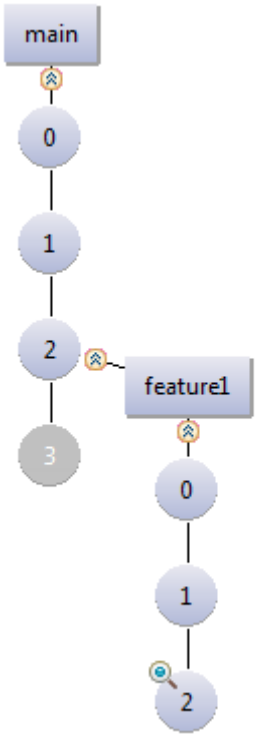
There is no command in the Model RealTime user interface for merging model files in the file system, because usually you don't have different versions of the same model in a project. But you can merge files in the file system from the command-line using the Command Line Tool for Compare/Merge.

Merge from CM System

When using a CM system, such as ClearCase or Git, it is usually easiest to merge two versions of a model by triggering the merge from the version tree where you can see all available versions of the model.

ClearCase

With ClearCase you use the view called ClearCase Version Tree. Make sure that your ClearCase view selects the version of the model that you want to modify (i.e. the target version for the merge). Then select the version you want to merge from and perform the command *Merge to View Selected Version* in the context menu. For example, consider the version tree shown below:



To merge the changes from version 3 on the main branch with version 2 on the feature branch, you right-click on version 3 and perform *Merge to View Selected Version*. ClearCase will then checkout version 2 on the feature branch and start the merge session. When the merge session is completed, the merge result model is committed to the checked out file. You can then check in the resulting model as version 3 on the feature branch.

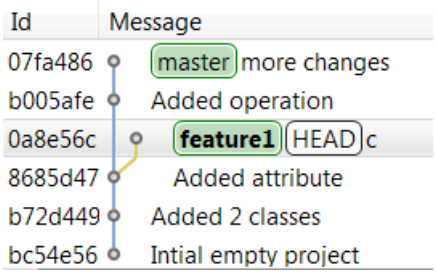
You can also trigger the merge from the command-line using tools such as "cleartool". For example, for ClearCase the equivalent of the above mentioned merge operation would be

```
cleartool findmerge HelloWorld.emx@@\main\feature1\2 -fta main_view -nc -
merge -gmerge
```

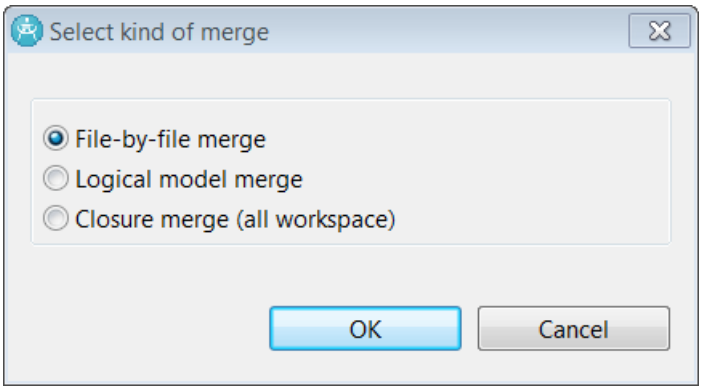
Here we just specify the target version of the model, and then use the `-fta` argument to specify the source view that selects the latest version on the main branch. ClearCase will attempt to do an automatic merge, but if this fails because there are merge conflicts that need to be resolved, the merge session will be sent to Model RealTime using the Compare/Merge Server. You then perform the merge using the Merge user interface in Model RealTime, and when you commit the merge result, ClearCase considers the file to be merged.

Git

With Git you use the History view configured to show all branches and tags. Make sure you have the target version for the merge checked out. Then select the branch you want to merge from and perform the command *Model Merge* in the context menu. For example, consider the version tree shown below:

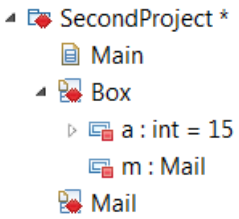


To merge the changes on the master branch with the branch called 'feature1', you right-click on the master branch commit (07fa486) and perform *Model Merge*. By default a dialog appears which lets you choose how to perform the merge (the preference *Team – RSx EGit Integration – Merge kind selection* can be set to avoid this dialog):

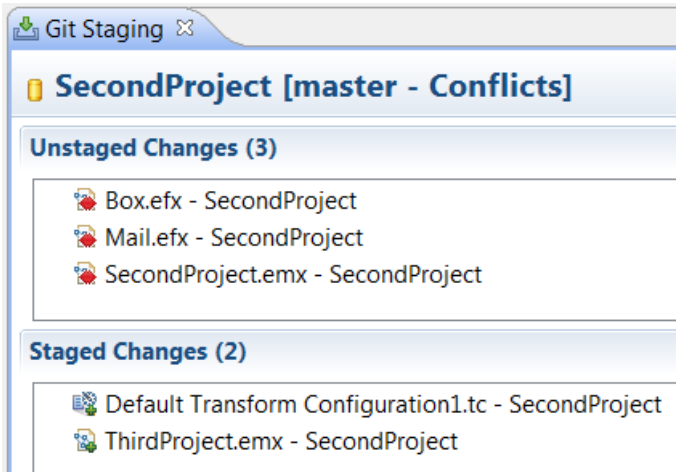


See Logical Models to learn about what logical model and closure merge means.

The merge is performed automatically, without showing the Merge user interface. You will be notified with a dialog if conflicts are detected and they are then shown using red icons in the Project Explorer on those top-level nodes that are stored in their own files:

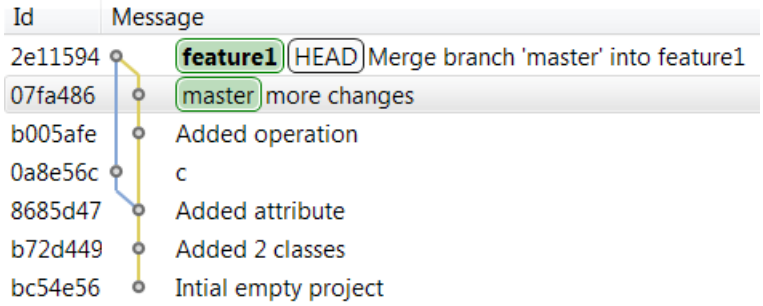


Files with conflicts are also shown in the Git Staging view under “Unstaged Changes”. Files that could be merged without conflicts will be automatically staged and will hence show up under “Staged Changes”.



To resolve conflicts you can right-click on a file marked to have conflicts in the Git Staging view and perform the command *Merge Tool*. It is also available from the Project Explorer context menu under the Team menu. The command brings up the Merge user interface which you then use for resolving the conflicts. When you have committed the merge session your workspace contains the merge result. By default files are automatically staged when you commit the merge session. There is a preference *Team – RSx Egit Integration – Automatically add resources to index after resolving conflicts* which you can disable if you prefer to manually stage the merged files.

When the merged files have been committed to the Git repository the History view will update to show that a merge has taken place:



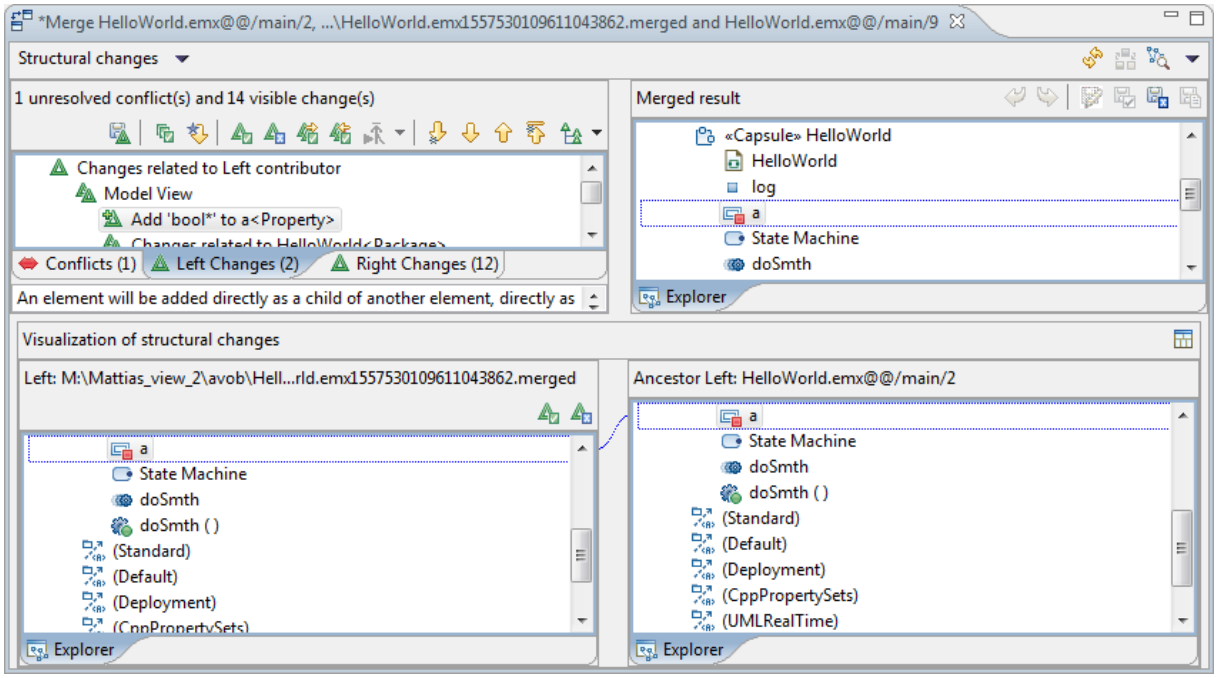
Note that if there are no merge conflicts, you cannot run the *Merge Tool* command and hence the Merge user interface will not appear. In this case you can instead run a Compare of the merge result model in your workspace with the predecessor version to ensure that it looks as expected before committing it to the Git repository.

It is possible to launch the Merge tool from the Git command line. You need to configure the git mergetool to invoke the Command Line Tool for Compare/Merge.

Merge User Interface

The Merge user interface is to a large extent the same as the Compare user interface (see Compare User Interface). However, some additions are made to support the more complex merge workflow.

The Merge user interface consists of four compartments:



The top left compartment shows the list of changes. They are organized into three tabs:

- **Left Changes**
  These are the changes between the left contributor and the common ancestor model.
- **Right Changes**
  These are the changes between the right contributor and the common ancestor model.
- **Conflicts**
  These are the changes that are in conflict and that need to be resolved during the merge session.

The bottom two compartments show one of the contributor models and the ancestor model. If you select a change from the Left Changes tab the left contributor model and the ancestor model will be shown. And if you select a change from the Right Changes tab the right contributor and the ancestor model will be shown. This is how you can understand what has happened to each contributor model since the common ancestor version.

If you press the button *Show All Contributor Panes* () you will see both the left and the right contributor panes at the same time as the common ancestor model. This may be useful when you have selected a conflict, to better understand how the changes from the left and right contributors are in conflict with each other.

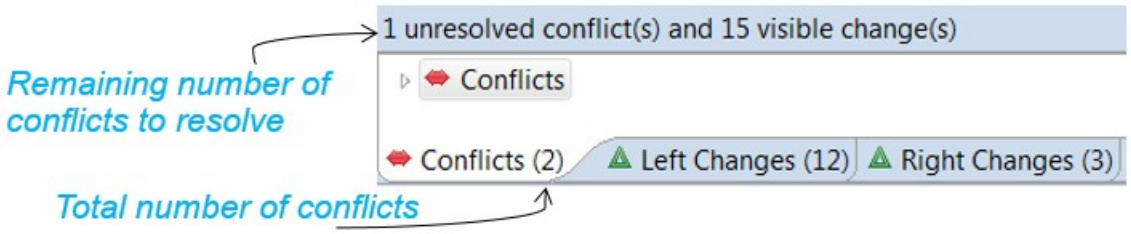
The top right compartment shows the merge result model. This is the model that you edit during the merge session. It is edited by accepting or rejecting changes from the left and right contributor models. When the merge session is completed, and you press the *Commit Merge Session*[1] button (), the merge result model will be saved. It is usually recommended to have the merge result model loaded in your workspace, because then you can continue to edit it di-

1 Git users should not be confused by the terminology here. Committing the merge session just means the merge session is completed. It does not mean that the merge result model is also committed to Git.

rectly after completing the merge session. Sometimes it is necessary to edit the merge result model manually after a merge session has been completed because some of the merge conflicts could not be properly resolved by the Merge tool. For example, you may want to resolve a merge conflict by accepting both the left and the right contributor change, and then make some small modification to the resulting model. The Merge tool won't let you do this, and such more advanced ways of resolving conflicts have to be done as a post-merge edit operation. You may use Compare/Merge Tasks to help remember changes that are required after completing the merge session.

Before we go into the details of the Merge user interface, here is a brief summary of the basic steps you take when merging two models:

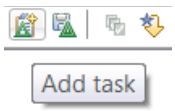
1. When the merge session starts, and the Merge user interface appears, all non-conflicting changes have by default already been accepted, and are hence present in the merge result model from start.
2. Click on the Conflicts tab in the change list compartment and check if there are any conflicting changes that need to be resolved. The number within parenthesis tells you how many conflicts there are, and the text above the change list tells how many of these conflicts that remain to be resolved. For example:



3. If there are unresolved conflicts select the first one from the list. Look at the visualization of the changes in the left and right contributor models using the bottom compartments. When you understand how the changes conflict with each other, decide if the conflict should be resolved by accepting the left change, the right change or none of the changes. See Resolving Conflicts for more information.
4. If you don't understand the conflict, or suspect that it may be a consequence of another conflict, skip the conflict and proceed to the next one. Work your way through all conflicts in the order in which they appear in the Conflicts tab.
5. When you see the message above the change list that there are 0 unresolved conflicts you are done, and can finish the merge session. This is done by pressing the "Commit Merge Session" button ( ).

Now let's take a closer look at the Merge user interface. We will only cover the merge-specific parts of the user interface. Most parts of the user interface is the same as when comparing models (see Compare User Interface).

Add Task



This button is the same as in the Compare user interface (see Add Task), except that during a merge session it creates a different kind of Compare/Merge task based on a selected conflict which has been resolved, or based on an element that is selected in the merge result model.

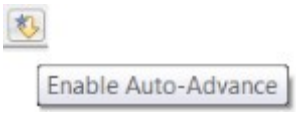
The created task can help you remember how a particular conflict was resolved, or that a certain element in the merge result model needs to be manually edited after completing the merge. See Compare/Merge Tasks for more information.

Accept All Non-Conflicting Changes

Pressing this button will accept all changes from the left and right contributor models that are not conflicting with each other. The changes will be applied to the merge result model. When you start a merge session all non-conflicting changes are by default automatically accepted, so you only need to use this button if you later have rejected some of these changes, and now want to accept them all again.

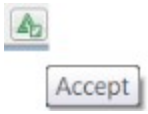
If you don't want the Merge tool to automatically accept non-conflicting changes when the merge session is started you can turn off the preference *General - Compare/Patch - Modeling Compare/Merge - Automatically accept resolvable differences for repository merge session*.

Enable Auto-Advance

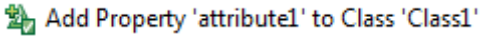


If this toggle button is pressed, the Merge user interface enters a mode where it automatically selects the next conflict as soon as you have resolved the previous one in the list.

Accept



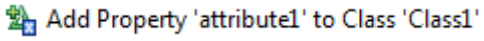
Pressing this button will accept the change that is selected in the change list. If the selected change is conflicting with another change, then accepting the change will automatically reject the conflicting change. When a change has been accepted the merge result model is modified so that it contains the change. Also, to show that the change has been accepted a small "accepted" decorator will be shown on the change. For example, the accepted change shown below means that the merge result model has been modified to now contain the added attribute "attribute1":



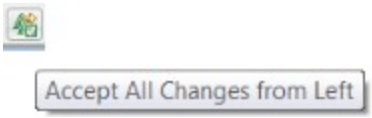
Reject

Pressing this button will reject the change that is selected in the change list. Note that if the selected change is conflicting with another change, then rejecting the change will not automatically accept the conflicting change. You have to specifically accept that conflicting change if you want to have it in the merge result model. When a change has been rejected, and the change previously existed in the merge result model, then the merge result model is modified so that it no longer contains the change. Also, to show that the change has been rejected a

small "rejected" decorator will be shown on the change. For example, the rejected change shown below means that the merge result model has been modified to now not contain the added attribute "attribute1":

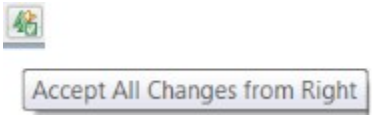


Accept All Changes from Left



Pressing this button will accept all changes from the left contributor model. This command is equivalent to selecting the changes in the “Left Changes” tab one by one, and do *Accept* on each of these changes. This means that all conflicts will be resolved afterwards.

Accept All Changes from Right



Pressing this button will accept all changes from the right contributor model. This command is equivalent to selecting the changes in the “Right Changes” tab one by one, and do *Accept* on each of these changes. This means that all conflicts will be resolved afterwards.

Next/First Unresolved Buttons

These buttons can be used to quickly go through the list of merge conflicts that remain to be resolved. When you select the first conflict in the Conflicts tab the *Next Unresolved* button becomes enabled and you can press it to go to the next unresolved conflict. And if you press the *First Unresolved* button the first of the conflicts that is still unresolved will be highlighted. Hence, these buttons make it convenient to find unresolved conflicts, especially for the situation when you choose not to resolve the conflicts in the same order that they appear in the “Conflicts” tab.

Undo/Redo Buttons

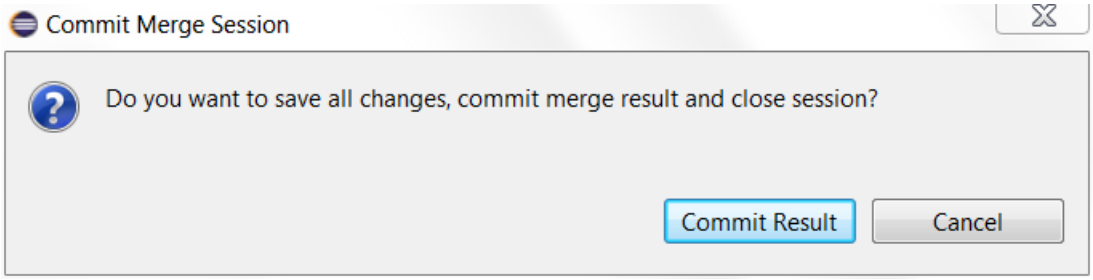
These buttons operate on the merge result model. You edit the merge result model by accepting or rejecting changes from the contributor models, or in the case of a textual sub merge, also by directly editing the merge result model. You can use the *Undo* and *Redo* buttons to undo such edit operations.

Commit Merge Session

This button becomes enabled as soon as you have resolved all the merge conflicts. Pressing it will complete the merge session, by committing the changes that have been made to the merge result model.

For Git users the term “commit merge session” may be slightly confusing, since this has nothing to do with Git commits. What is meant by committing a merge session is to complete it and save all results in the merge result model. The changes will not be committed to Git.

Before the merge session is committed you will be prompted for confirmation:

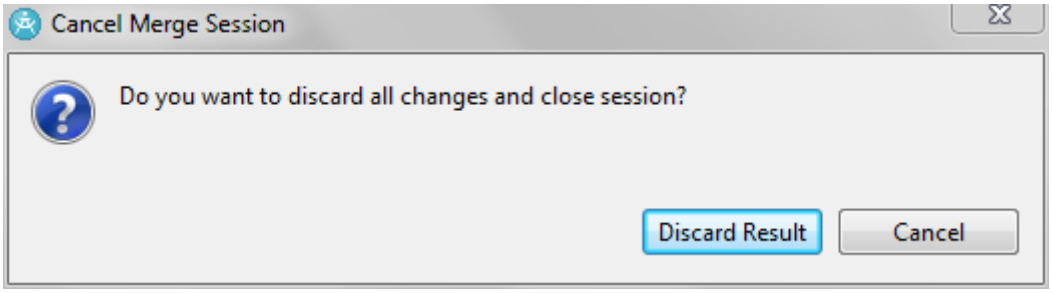


Press *Cancel* to return to the merge session and *Commit Result* to finish the merge session and save the merge result model to disk. After the merge session has been committed you can continue to edit the merge result model just like any other model that you have in your workspace. For example, you may need to do post-merge editing to fix issues caused by conflicts that could not be properly resolved using the Merge user interface.

Another way to commit the merge result and finish the merge session is to simply close the Merge editor window, and then answer Yes to the question about saving the merge result model. However, this approach is not recommended because if you missed to resolve any of the conflicts, the saving of the merge result model will be aborted, and in this case you will loose all the changes you have done during the merge session.

Cancel Merge Session

If you press this button the merge session will be cancelled. All changes you have made to the merge result model by accepting and rejecting changes, and by editing in textual sub merge sessions, will be lost. Before the merge session is cancelled you will be prompted for confirmation:



Press *Cancel* to return to the merge session and *Discard Result* to stop the merge session and discard the merge result model.

When using ClearCase note that files that were automatically checked out when the merge session was started will still be checked out after the merge session has been cancelled, and you may want to do *Undo Checkout* on these files.

You can also cancel the merge session by simply closing the Merge editor window, and answer No to the question about saving the merge result model.

Save a Copy

Pressing this button will save a copy of the merge result model. You can only do this when all merge conflicts have been resolved. The main reason for saving a copy of the merge result model is to remember what the model looked like immediately after the merge session was completed, before starting to do any post-merge editing. If you want to save the merge result before the merge session is completed, you first have to ignore all remaining unresolved conflicts. This may be useful if the merge is complex, and you want to save the merge result model at a particular point in time before certain complex conflicts have been resolved.

It can also be useful to have a copy of the merge result model if you later find that the merge session introduced errors in the model, for example because conflicts were resolved in an inappropriate way. You may then want to redo the complete merge session, and resolving the conflicts in a different way. A copy of the previous merge result model can then be useful to avoid making the same mistake again. Also, you can of course compare saved merge result models using the Compare tool, to study the effect of resolving conflicts in a different way.

If you perform the *Save* or *Save All* command when the Merge editor is open, the merge result model will be saved to a temporary location. However, saving is only possible if there are no unresolved merge conflicts.

Resolving Conflicts

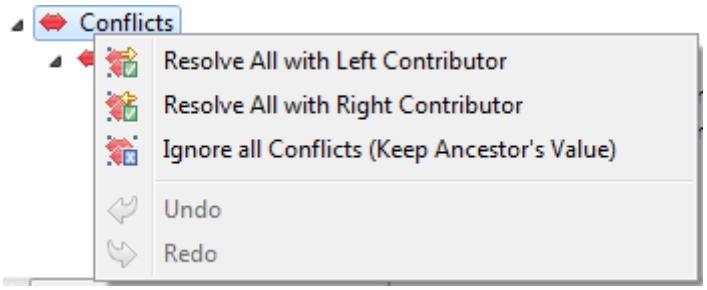
The time it takes to perform a merge session very much depends on the number of merge conflicts you have to resolve. If there are no, or few, conflicts the merge will usually be quick and easy, while if the number of conflicts is big, the merge will take a longer time to complete and may be more complex. Therefore, it is important to work in a way that reduces the number of conflicts you get when merging models.

The main advice for increasing the likelihood of getting simple merges is to merge frequently. If you are working on a feature stream, it is a good idea to compare your version of the model on this feature stream with the latest version on the main stream at regular intervals. If you detect changes you should consider rebasing, to merge these changes into your feature stream sooner rather than later. For the same reason it is important that you deliver your changes to the main stream incrementally, and avoid waiting a long time and then delivering a big amount of changes in one big delivery. Following this approach will help developers that work on other feature streams so that they don't get too many conflicts the next time they rebase their features streams.

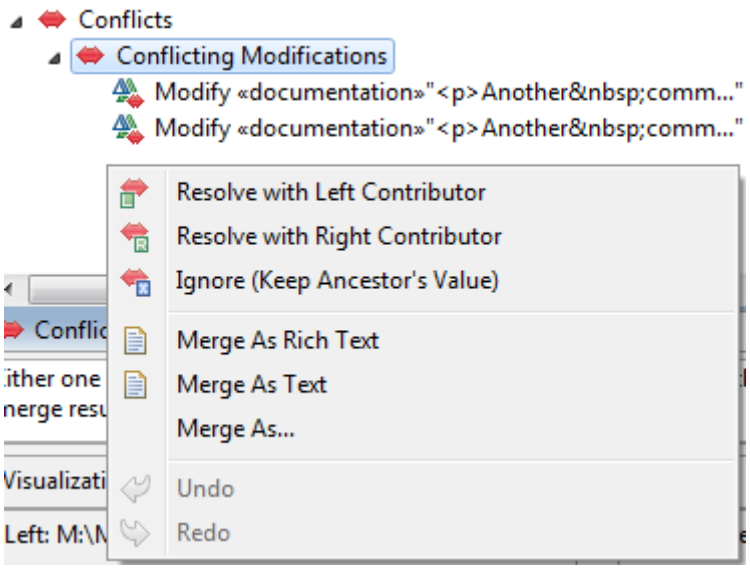
Of course, the importance of following the above recommendations becomes bigger if many feature streams make changes to the same parts of the model. For features that are more isolated from the work of others, you don't have to rebase and deliver that frequently since you won't get many conflicts anyway.

If there are merge conflicts reported you have to resolve these before you can complete the merge session and commit the merge result. Use the Conflicts tab and the *Next Unresolved* button to go through the merge conflicts one by one. Model RealTime by default sorts the conflicts so that the ones that are best to resolve first appears at the top of the list. For example, conflicts related to model elements appear before conflicts related to diagrams, since resolving a conflict for a model element often automatically resolves a diagram related conflict too.

In some special cases you may want to resolve all conflicts by consistently choosing the change from either the left or right contributor. For example, if you only did very few changes on the feature stream, but still got lots of conflicts, it may be easier to resolve the conflicts by accepting all the changes from the main stream, and then redo your feature stream changes again after the merge has been completed. To resolve all conflicts right-click on the top-level Conflicts node in the Conflicts tab and choose either *Resolve All with Left Contributor* or *Resolve All with Right Contributor*. You can also choose to resolve the conflicts by ignoring them, that is to neither accept the left nor the right changes. In that case the affected model elements will look like in the common ancestor model once the merge is completed.



However, the most common scenario is to resolve conflicts individually, one by one. Each conflict is represented by a separate node in the Conflicts list, and below this node you can see the changes from the left and right contributor that is in conflict with each other. To resolve a conflict right-click on its node in the Conflicts tab:



Choose *Resolve with Left Contributor* to accept the change from the left contributor and at the same time reject the change from the right contributor. Or choose *Resolve with Right Contributor* to do the opposite. You can also choose to resolve the conflict by ignoring it, that is to reject both the left and right change. This can be useful if the conflict needs to be resolved by somehow combining the left and right changes. The Merge tool won't let you do this, but you can resolve the conflict by ignoring it, and then later when the merge session is completed, return to the affected model element and edit it manually. It's a good idea to keep a separate list of elements that you need to get back to once the merge session is completed. The Compare/Merge Tasks view can help you with this.

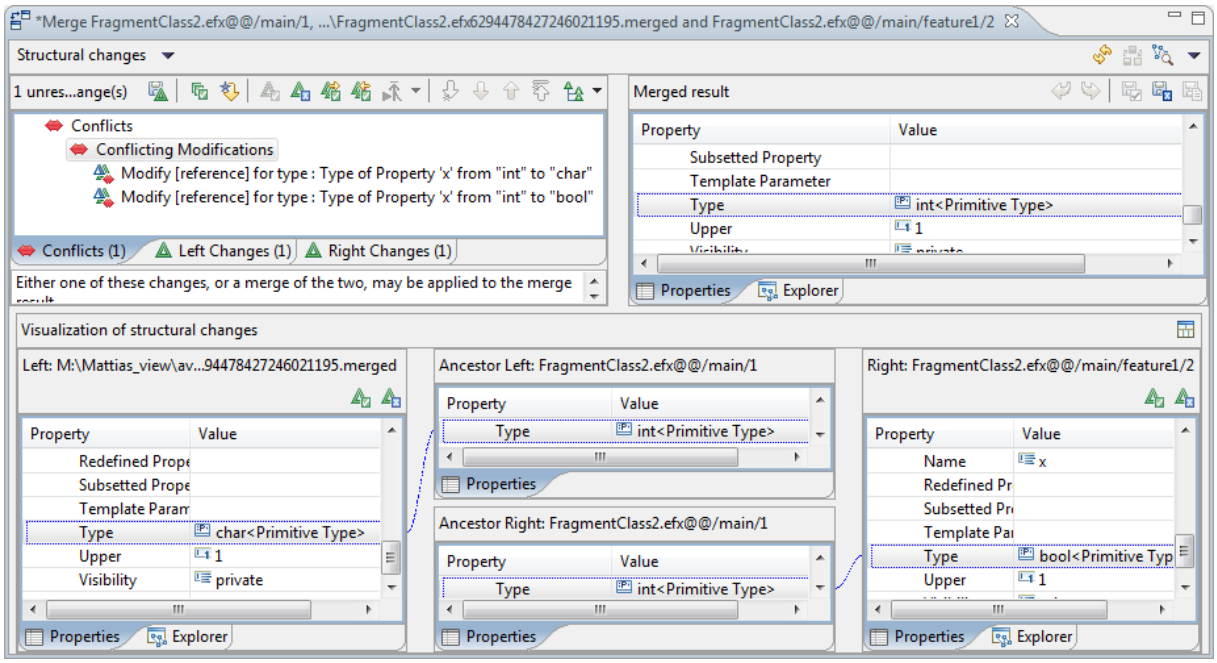
As you can see in the above picture, conflicts related to textual modifications can also be resolved by invoking a sub merge session. See Text Merge for more information about this possibility.

Before you can decide how to best resolve a particular conflict you usually have to study the conflicting changes carefully. The name of the conflict describes the type of conflict and in which way the left and right changes are conflicting. Here are some examples of common kinds of conflicts you may encounter:

- **Conflicting Modifications**
  The same element property was modified in both contributors.
- **Conflicting Addition and Deletion**
  An element was added in one contributor, but one of its container elements was at the same time deleted in the other contributor.
- **Conflicting Modification and Deletion**
  An element was modified in one contributor, but at the same time deleted in the other contributor.

There are many more possible combinations in which two changes may conflict. See Types of Changes for a list of all the possible changes which may be part of a conflict.

When a conflict is selected the Merge user interface shows in the bottom compartments how the element has been modified in both the left and right contributor models compared to the common ancestor model. Let's look at an example:



Here we have a conflict that is caused by the type of an attribute "x" being changed in both contributor models. In the bottom compartments we can see that in the left contributor the attribute's type was changed from "int" to "char", while in the right contributor it was changed from "int" to "bool". You can also select one of the changes shown below a conflict node to only look at the details of that particular change in one of the contributors. Then the four compartments shown in the picture above will be replaced by only two compartments; one for the contributor model and one for the common ancestor model. Thereby the compartments can take up more space, which in turn can make it easier for you to see exactly what has changed.

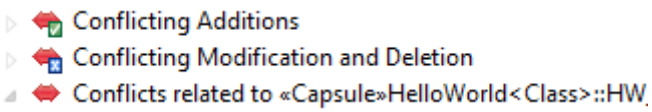
As you can see in the picture above there are *Accept* and *Reject* buttons also in the compartments that show a change in a contributor model. These buttons work in the following way:

- If you press the *Accept* button the change from that contributor will be accepted. At the same time the conflicting change from the other contributor will be rejected.
- If you press the *Reject* button the change from that contributor will be rejected. However, this does not mean that the conflicting change from the other contributor gets accepted automatically. Instead, the merge result becomes the same as the common ancestor model, and you have to explicitly press the *Accept* button for the other contributor if you want that change to be in the merge result model.

If you encounter a merge conflict that is hard to understand, a pragmatic approach is to simply skip it and proceed to the next conflict. Since resolving one conflict may automatically also resolve other conflicts there is a chance that the skipped conflict will disappear once you have resolved another conflict in the list.

Another guideline is to first resolve those conflicts that have a big impact on the merge result model, such as adding or deleting container elements in the model, before looking at conflicts for elements that are contained within these elements. When you resolve such "container element" conflicts it is likely that many other conflicts will be automatically resolved too.

Conflicts that have been resolved are marked with an icon to distinguish them from unresolved conflicts. For example:

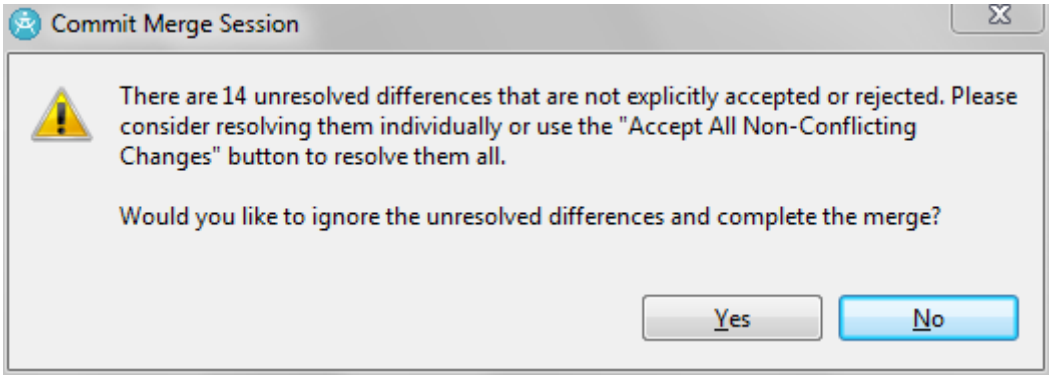


Here, the first conflict has been resolved by accepting either the left or right change, the second conflict has been resolved by ignoring it (i.e. neither the left nor the right change was accepted), and the third conflict is still unresolved.

Dependent Changes

A model consists of elements that are highly interconnected. Therefore it is not uncommon that changes depend on each other. That is the reason why resolving one conflict may automatically resolve other conflicts at the same time. When you accept or reject changes the Merge tool will evaluate the effect those changes will have on the merge result model. The overall requirement is that the merge result model must never be inconsistent or corrupted by the merge. Dependent changes are those changes that the Merge tool has to automatically accept or reject depending on the choices you make during the merge session.

Sometimes you may encounter a situation where you have resolved all conflicts, but still there may be dependent changes from either the left or right contributor models that are neither accepted nor rejected. Because these are dependent changes the button *Accept All Non-Conflicting Changes* can not always be used to accept such changes. If you try to commit the merge result and finalize the merge session when you still have such dependent changes that are neither explicitly accepted nor rejected you will get a warning dialog:



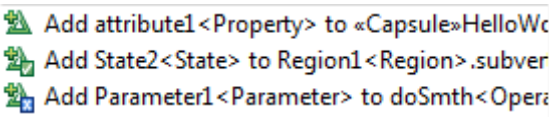
You may proceed and complete the merge, but then these changes will be ignored and will hence not be present in the merge result model after the merge. It is strongly recommended that you use the Left and Right tabs to look at these changes and make an explicit decision for each of them whether to accept or reject them. To know whether the unresolved changes come from the left or right contributor model you may click on the top-level nodes in the Left and Right tabs. This shows a message in the text area[2] below that tells you how many changes there are in total for that particular contributor, and how many of these changes that are cur-

2 This text area is by default hidden but can be made visible by running the command *Show description section* in the Delta Tree Configuration button menu.

rently unresolved. For example, for the situation that caused the warning dialog shown above you may see the following message for one of the contributors:

There are 49 change(s) in the contributor. Among them, 14 change(s) still require resolution.

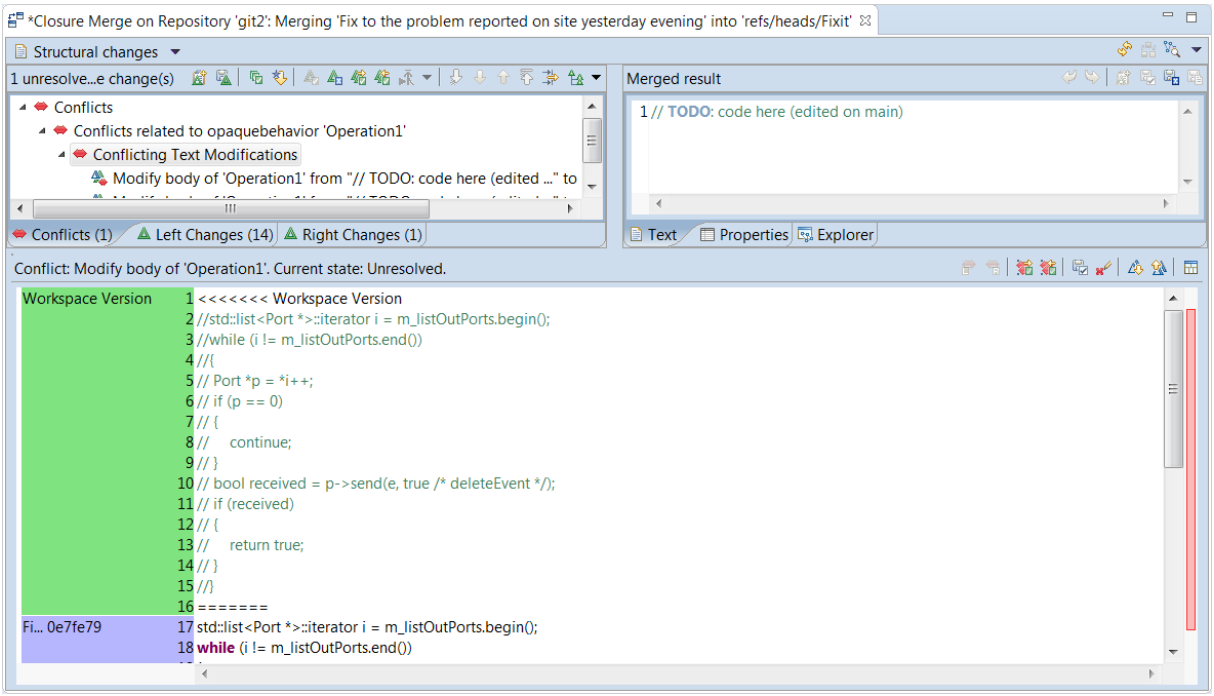
When you go through the contributor changes you should look for changes that are not marked as neither accepted nor rejected. For example, in the picture below the first change has neither been explicitly accepted nor rejected, while the second has been accepted and the third has been rejected.



Text Merge

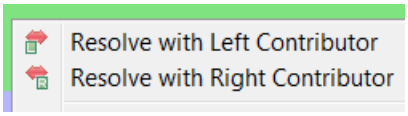
If you get a merge conflict because both the left and right change involves a property that contains a piece of modified text, such as a code snippet, then you resolve the conflict by performing a textual merge of the modified texts.

Here is an example of what the Merge editor could look like when you select a conflicting text modification:



In the text editor at the bottom you see the merged text. For conflicting parts the left and right versions are highlighted with colors (customizable in the preferences at *General – Compare/Patch – Modeling Compare/Patch – UML Compare/Merge – Text annotation settings*). There are also markers in the text showing which contributor a particular text fragment comes from. The names of the versions (Git commit messages in the above picture) are shown in the left column. If the version texts are too long to fit, hover the cursor over them to see the full texts in a tooltip.

You can resolve each conflict by right-clicking in the text editor and perform one of the following commands:



If you want to resolve all conflicts in the text in the same way you can use the corresponding toolbar buttons:

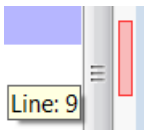
Initially the "Merged result" compartment in the upper right corner shows the ancestor version of the text. But as soon as you save your changes in the text editor, it gets updated with the contents of the text editor. That is, what you see in the text editor is really the text that will be committed to the model (when you save).

Hint: Find/Replace can be used in both the text editor area and also in the "Merged result" compartment.

In addition to using the commands for resolving conflicts, you can also resolve them by simply editing the text directly. This can be very useful if the left and right versions somehow need to be combined, or if you want to resolve a conflict with a completely different piece of text. You can make edits anywhere in the text, not only in conflicting parts.

The information text above the text editor shows the current state of the text merge. Initially it says "Unresolved" (or "Auto resolved" in case there were no conflicts in the text), but as soon as you have saved, so that the Merged result is updated, it will say "Manually merged". You can use the command *Discard edit changes* () to revert your changes and get back to the "Unresolved" state.

Use the *Next/Previous Difference* buttons () to walk through all conflicting parts in the text. You can also use the red markers in the right editor margin for quickly navigating to the conflicts. The tooltip of these markers show the line where the conflicting piece of text is located:

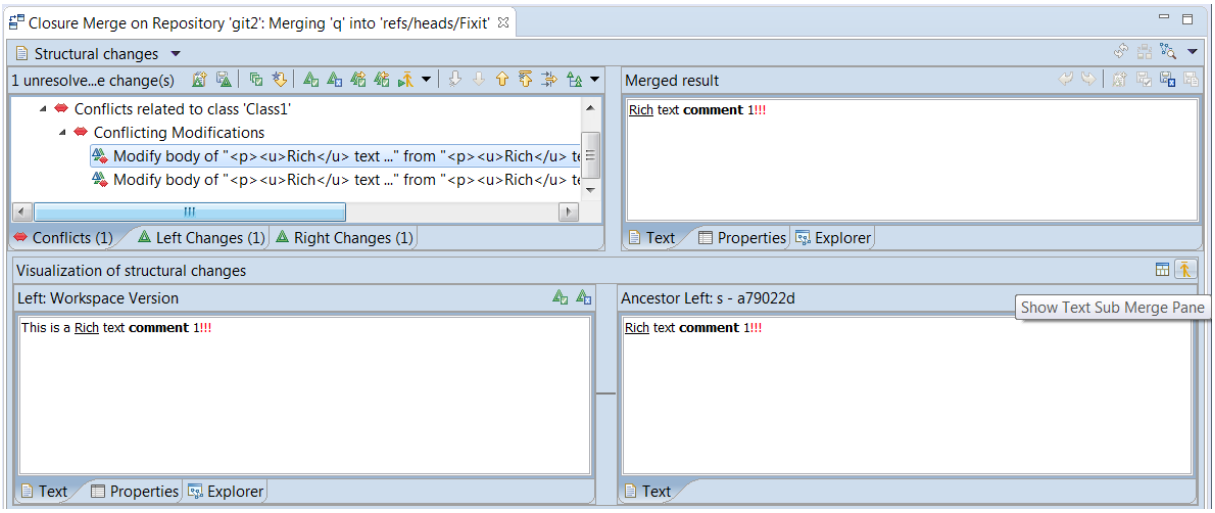


Press the *Toggle Compare/Merge View* button () to hide the text editor and instead show a comparison between the ancestor version and the current merge result. This gives you an overview of all changes you have made in the text during the merge session, and it can therefore be useful in order to review the merge changes as a last step in resolving the conflict.

Merging Rich-Text Comments

Merge of rich-text comments cannot be done in the same way as merging other pieces of text. This is because rich text comments may contain annotations (markup), such as font styling, images, tables etc. Allowing the merge result to be edited freely could in this case lead to cor-

rupted rich text that would not render correctly. Therefore the user interface for rich text merging looks differently, and does not allow free editing of the merge result.



You have to resolve conflicts in rich text the usual way by accepting either the left or right version (or none). By pressing the button *Show Text Sub Merge Pane* you can invoke Sub Compare in order to get more information about the conflict. If you want more flexibility when resolving conflicts in rich text, it's recommended to edit the model after the merge. See Compare/Merge Tasks for how to track needed post-merge changes during a merge session.

Structural Merging by Combining Models

When the Merge tool merges two models it by default uses the unique identifiers of the model elements to compare the contributor models with the common ancestor model. If an element with the same unique identifier is found in both the common ancestor model and in one of the contributor models, the Merge tool knows that this element has not been added in the contributor model.

One consequence of matching elements by means of their unique identifiers is that if two developers, that work on different streams, add a class with the same name in the same package, then these classes will have different unique identifiers and hence will be treated as different classes by the Merge tool. However, since Merge knows that it is not allowed in UML to have two classes with the same name in a package, it will report an Add/Add conflict for this case, although strictly this is not a structural conflict but more of a logical conflict.

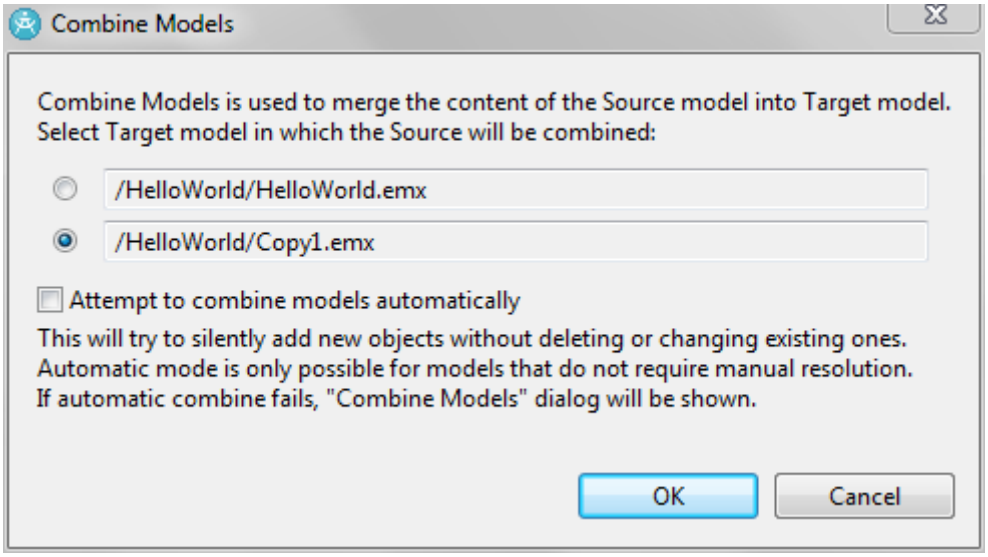
So when merging you have to choose one of the classes, but not both. But what if the two developers really meant that these two classes should be the same class, based on the fact that they gave the classes the same name? The wanted merge result would then be just a single class which combines the contents of the classes from both contributor models. In situations like this one you need to merge models structurally, without using their unique identifiers as the matching criteria.

Model RealTime supports a feature called Combine Models. It takes two models and performs a structural merge of them, so that one of the models (the source model) is combined into the other model (the target model). The source and target models can be any models and do not

need to be versions of the same model. That is, there is no common ancestor model involved when merging models using the Combine Models feature.

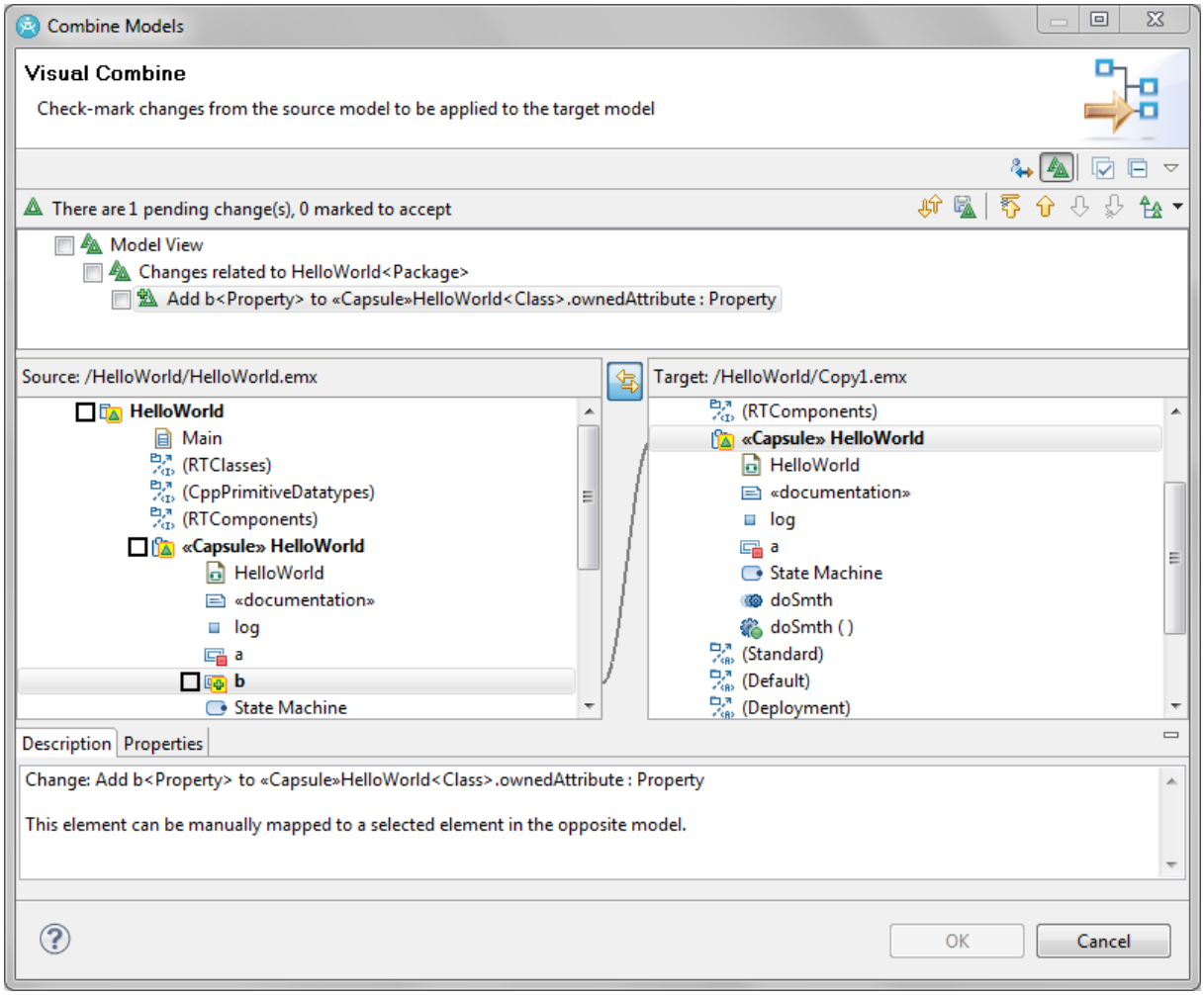
Before you combine two models it is a good idea to take a copy of the target model. This is because Combine Models will update the target model directly, giving you no possibility to roll-back the changes unless you have a backup copy.

To use the Combine Models feature select two elements in the Project Explorer that are stored as the root in a model file, right-click and choose the command *Combine Models* in the context menu. A dialog appears where you can specify which of the selected models that should be the target model:



Unless you are sure about the result of combining the models, it is recommended to leave the checkbox "Attempt to combine models automatically" unmarked. Combining models automatically is similar to an automatic merge; it may be convenient and quick, but you may also get changes in the target model that you didn't expect.

The Combine Models user interface looks slightly different than the Merge user interface:



Combine Models match elements in the source and target models by name and structure rather than using their unique identifiers. In the example shown in the picture above the attribute "a" was added independently in both models. A regular merge would hence treat these as different attributes. But Combine Models consider them to be the same since they have the same name and structure.

To perform the combining of the models you mark all changes in the change list (the upper compartment) that you want to apply to the target model. You can also select the elements in the source model compartment. Finally press the OK button to commit the changes.

Most of the Combine Models user interface is the same as the Compare/Merge user interface. The main difference is the button *Show Manual Mappings Pane* ( ) which is explained below.

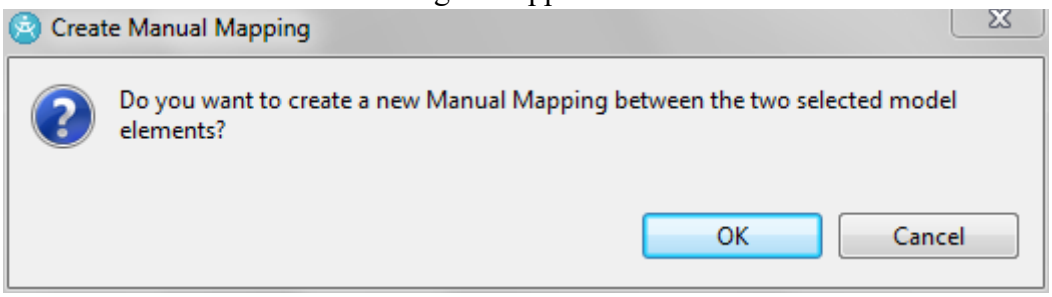
Manual Mappings

Sometimes two elements in the source and target models may have different names and/or structure, but still you want them to be matched as the same element when combining the models. In this case you have to set-up a manual mapping between these elements. This is done in the “Manual Mappings” pane at the top of the “Combine Models” dialog.

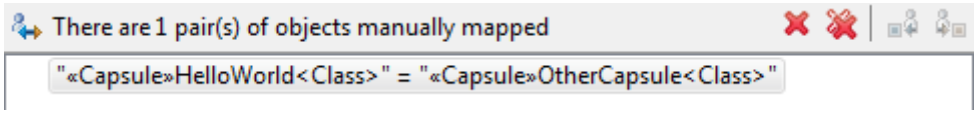
For example, assume we'd like to combine two capsules with different names:

To create a manual mapping between these two capsules, follow these steps:

1. Right-click on the "HelloWorld" capsule and perform *Select Source element for manual map*.
2. Right-click on the "OtherCapsule" capsule and perform *Select Target element for manual map*.
3. Press OK in the confirmation dialog that appears:



Now you can see the manual mapping you just created in the manual mappings pane:



Also, a blue line between the source and target model compartments shows that the two capsules now are mapped:

Now you can mark the checkbox for "HelloWorld" and press OK. Afterwards you will see that "OtherCapsule" now contains the port and the attribute from "HelloWorld". Also, the state machines were combined without the need to define a manual mapping, since they already had the same name.

Exploring the Contributor Models

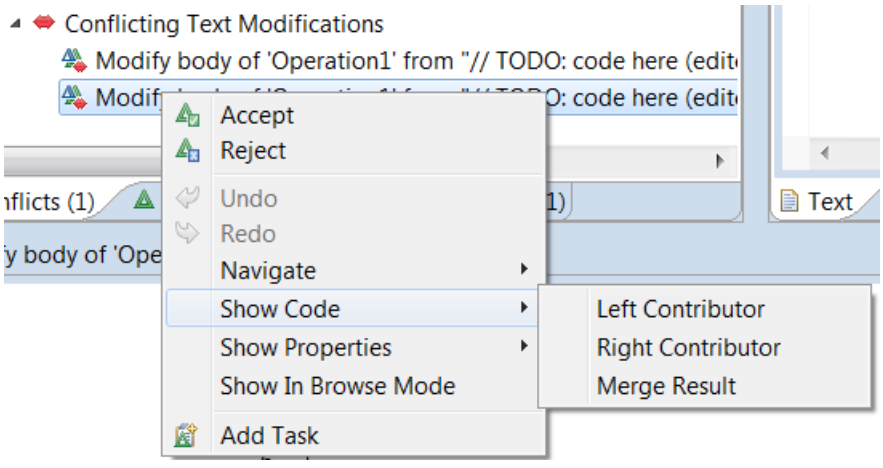
To get a better understanding of a change or merge conflict it is often necessary to look at the model context in which the changed model element is located. You may want to explore the changed model element to reveal more details about it than what is shown in the change description. You may also want to explore other related model elements in either of the contributor models.

As already explained in Browse Button you can use the browse mode of the Compare/Merge tool to explore the contributor models in more detail using the Explorer, Properties and Diagram tabs of the contributor model compartments of the Compare/Merge user interface. How-

ever, you can also use some of the regular Model RealTime views that are updated when an element is selected in the Compare/Merge user interface:

- **Properties view**
  The Advanced tab of the Properties view is similar to the Properties tab that appears in the browse mode. But the Properties view also provides several other tabs that make it easier to find the property you are looking for. When showing properties for an element in a contributor model the Properties view behaves in the same way as when it shows properties for an element in the workspace model, except that it does not allow you to edit any property. Navigation by means of hyperlinks is also disabled.
- **Code view**
  You can use the Code view to look at code snippets for elements that are selected in the contributor models, or in the merge result model. Just like the Properties view, the Code view will be read-only when it shows such a code snippet. However, you can use the navigation commands that are provided by the Code view, for example in order to navigate to the Project Explorer or a diagram. Just remember that when you open a diagram from one of the contributor models or the merge result model, that diagram will be read-only as well.

When you select a change or conflict in the change list compartment, the Properties view and Code view are also updated. In this case the changed model element in the left or right contributor models is used (depending on if the change is located in the Left or Right tab). If the changed model element does not exist in that particular contributor model (as will be the case for a Delete change), no information will be shown in the Properties view or Code view. If you right-click on the change or conflict you can also choose to look at the changed model element in the other contributor model, the common ancestor model or the merge result model. Depending on the kind of change and whether you are doing a compare or merge session not all these models may be available to choose from in the context menu.



Types of Changes

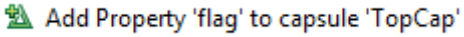
Depending on how the contributor models have been modified a compare/merge session may show different kinds of changes. Below we go through all the different types of changes you may encounter when comparing or merging two models.

Add Change

An Add change means that a new element has been added to the left contributor model. The change description is on the form

```
Add <element> to <location>
```

where <element> describes the new element and <location> describes where in the model it has been added.



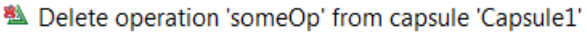
For example, the change shown above means that an attribute (a.k.a Property) "flag" has been added to the capsule "TopCap".

Delete Change

A Delete change means that an element has been deleted from the left contributor model. The change description is on the form

```
Delete <element> from <location>
```

where <element> describes the deleted element and <location> describes from where in the model it has been deleted.



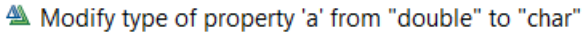
For example, the change shown above means that an operation "someOp" was previously contained in the capsule "Capsule1", but has now been deleted from the model.

Modify Change

A Modify change means that an element has been modified, so that one of its properties has changed in the left contributor model compared to what it is in the right contributor model. The change description is on the form

```
Modify <property> of <element> from <value before> to <value after>
```

where <element> describes the modified element, <property> specifies which property of the element that was changed and <value before> and <value after> describes the value of the property before and after the change.



For example, the change shown above means that the type of the attribute "a" was changed from "double" to "char".

Move Change

A Move change means that an element has been moved within a model. There are two ways in which an element can be moved. Either it is moved within the same owner element, or it is moved to a different owner element.
In the first case the element is reordered within its container collection, so that it still has the same owner element after the move. In this case the change description is on the form

```
Reorder <element> within <owner>
```

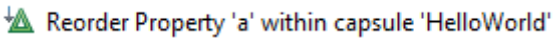
where <element> describes the moved element and <owner> is the owner element within which the element has been moved.

In the second case the element is moved so that it has a different owner element afterwards. In this case the change description is on the form

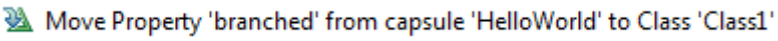
```
Move <element> from <old owner> to <new owner>
```

where <element> describes the moved element, <old owner> is the owner element before the move, and <new owner> is the owner element after the move.

Let's look at two examples:



This change means that the property (i.e. attribute) "a" was reordered within the capsule "HelloWorld". That is, after the move the capsule still contains attribute "a" but at a different position in its list of attributes.



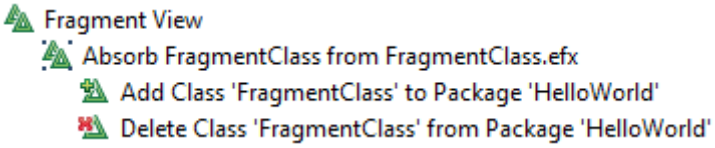
This change means that the attribute "branched" was moved from the capsule "HelloWorld" to the class "Class1".

Fragment Absorb Change

This change means that an element was absorbed from a fragment file (.efx), to instead be stored in a model file (.emx). The change description is on the form

```
Absorb <element> from <fragment file>
```

where <element> describes the element that was absorbed from the fragment file and <fragment file> is the affected fragment file that no longer stores the element afterwards.



For example, the change above means that the class "FragmentClass" was previously stored in the fragment file "FragmentClass.efx" but was absorbed into the model file instead. Below the fragment absorb change you can see that this change in fact is implemented by first deleting the class, so that it gets removed from the fragment file, followed by adding it again into the model file that now will store it instead.

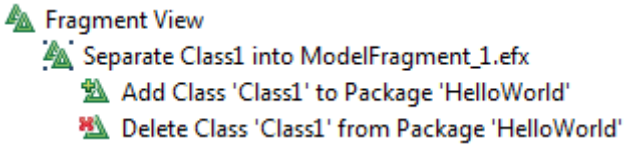
Note that the model does not change because of a fragment absorb change. The class is deleted from the same location in the model at which it later is added again. The only difference is in how the model is stored in files.

Fragment Separation Change

This change means that an element was separated out into a model fragment file (.efx). The change description is on the form

```
Separate <element> into <fragment file>
```

where <element> describes the element that was separated out to a fragment and <fragment file> is the fragment file that stores the element afterwards.



For example, the change above means that the class "Class1" is now stored in the fragment file "ModelFragment_1.efx". Below the fragment separation change you can see that this change in fact is implemented by first deleting the class, so that it gets removed from its previous file, followed by adding it again into the fragment file.

Note that the model does not change because of a fragment separation change. The class is deleted from the same location in the model at which it later is added again. The only difference is in how the model is stored in files.

Diagram View Position and Size Change

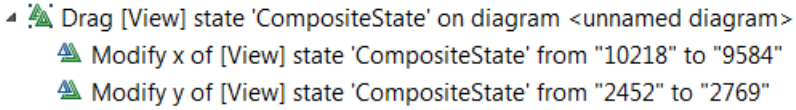
A change of this type means that a view (i.e. a symbol or a line) on a diagram has been repositioned or resized. The change description has the following form for a symbol position change:

```
Drag [View] <element> on diagram <diagram>
```

and the following form for a symbol size change:

```
Change Size of [View] <element> on diagram <diagram>
```

where <element> is the element whose view was repositioned or resized and <diagram> is the diagram where the view is located.

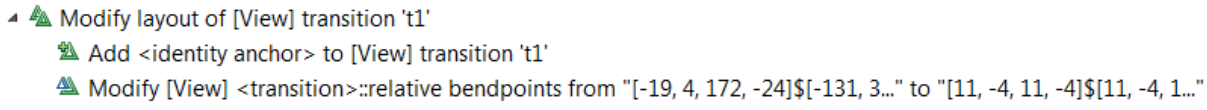


For example, the change above means that a state symbol showing the state "CompositeState" was moved on a state machine diagram. The details of how it was moved can be seen in Modify changes for the X and Y coordinates that are nested below this change.

When the layout of a line has changed, the change description has the following form:

```
Modify layout of [View] <element>
```

where <element> is the element whose line was rerouted.



For example, the change above means that a line for transition ”t1” has been rerouted. The detailed changes that are nested below it give some hints on how the line was rerouted, but usually it’s easier to see this visually in the Diagram tab.

Compare/Merge Induced Reference Modifications

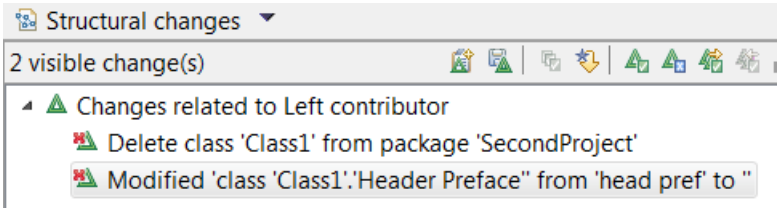
This type of change is used for changes that are not caused by any modification you have made to the model yourself, but that the Compare/Merge tool itself has induced when loading the contributor models. You may see such changes if you compare or merge models that have been created in older versions of Model RealTime. They will only appear the first time you compare or merge them. Once the models have been saved in the new Model RealTime version these changes will no longer appear in future compare/merge sessions.

One example of this kind of change is when additional information has been added in model files in order for Compare/Merge to be able to present changes in a better way. For example, in order to be able to show the name of the event that is referenced by a transition trigger that has been modified, it is necessary that the event name is available in the model file where the transition trigger is stored. This is because Compare/Merge may only get that single model file to operate on, and will then not be able to find out the name of the event unless it is present in that model file. Because of this recent versions of Model RealTime now stores the event name as an additional piece of information for transition triggers. However, the first time you compare or merge against a file that was created in an old version of Model Real-Time where this information was not present in the file, you will get a Compare/Merge Induced Reference Modification. There is a filter available in the Delta Tree Configuration button menu that you can use for hiding these kinds of changes.

Compare/Merge Induced Deletions

This type of change is used for elements that get indirectly deleted from the model when another element gets deleted, even if they are not contained within it. For example, when you delete an element which has stereotypes applied, the stereotype instances will be deleted too. However, since the stereotype instances are stored separately from the element to which they are applied, these deletions are not a structual consequence of the element deletion, but more of a logical consequence (i.e. it usually does not make sense to keep such stereotype applications once the element has been deleted).

Here is an example:



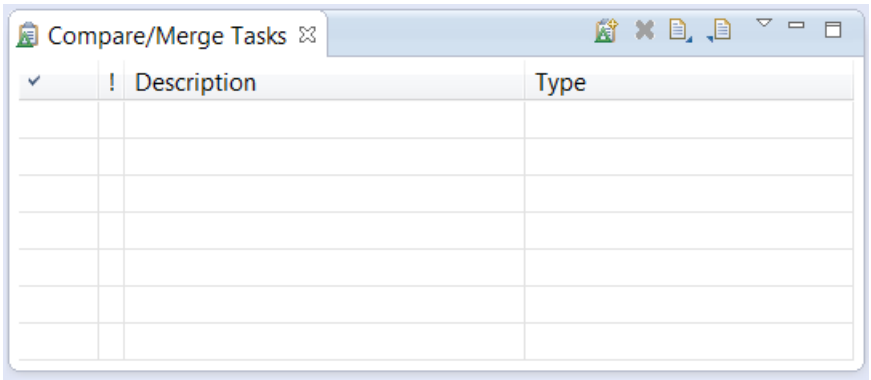
The ”Header Preface” code snippet for a class is stored by means of a stereotype. When the class is deleted, so is the code snippet.

There is a filter available in the Delta Tree Configuration button menu that you can use for hiding these kinds of changes.

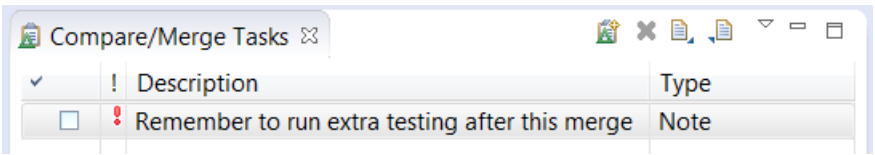
Compare/Merge Tasks

Model RealTime provides a dedicated view that allows you to keep track of notes taken during a compare or merge session. It can for example be used to remember actions that need to be performed after a merge session, or to write review comments for changes.

To make this view visible invoke *Window – Show View – Other* and then type “task” in the filter box to find the view called “Compare/Merge Tasks” (it's located in the “Compare/Merge” category).



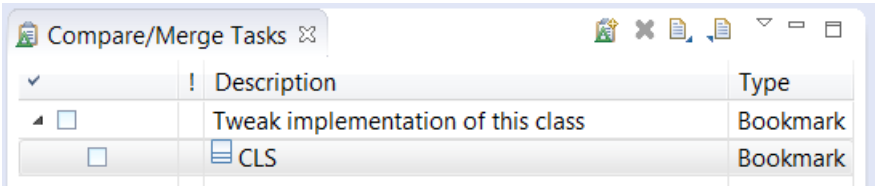
There are a few different kinds of Compare/Merge tasks. The most general one is a Note task. You create a Note task by pressing the *Add New Compare/Merge Task* button in the view's toolbar. Provide a description and optionally a more detailed text in the dialog that appears. Here is an example of a Note task created during a merge session.



Just like for tasks in the standard Eclipse Tasks view you may set a priority for a Compare/Merge task (Normal, High or Low). There is also a checkbox which can be used to mark the task as completed.

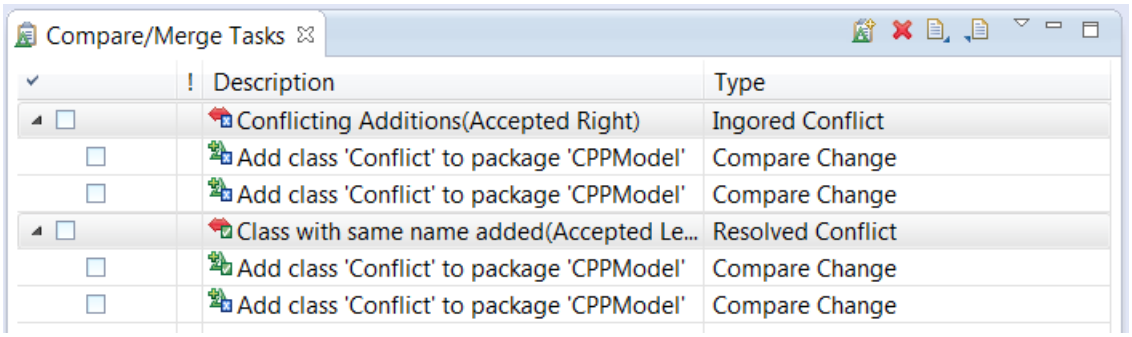
More specific kinds of Compare/Merge tasks are created using the Compare/Merge user interface:

- **Bookmark Task**
  The purpose of a Bookmark task is to set a “bookmark” on an element in a merge result model, so that you can easily find that element after the merge session is completed. For example, you may set a Bookmark task on an element which you think may require some post-merge editing.
  A Bookmark task is created by selecting one or many elements in the merge result model and then pressing the *Add Task* button () in the Compare/Merge editor toolbar. The command is also available in the context menu.



- The bookmarked element (or elements) appears below the bookmark task and you can double-click to navigate to it in the workspace model.

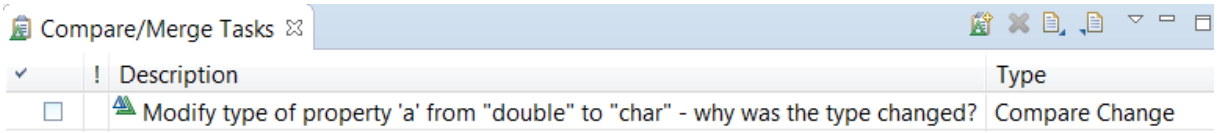
- **Ignored or Resolved Conflict Tasks**
  During merge a conflict can only be resolved by either accepting one of the contributor versions, or by ignoring them both. Sometimes this is not flexible enough, and you may want to resolve a conflict by taking parts from both contributors, or to take nothing from the contributors but instead manually change the merged element in some way.
  In these cases it is necessary to edit the merged model after completing the merge session. To remember where and what to edit, mark the ignored or resolved conflict and press the *Add Task* button () in the Compare/Merge editor toolbar (or use the context menu).



  These Ignored or Resolved Conflict tasks show how the left and right contributor changes were conflicting, and how you decided to resolve the conflict. You can also navigate to the involved elements in the workspace model by double-clicking on the tasks.

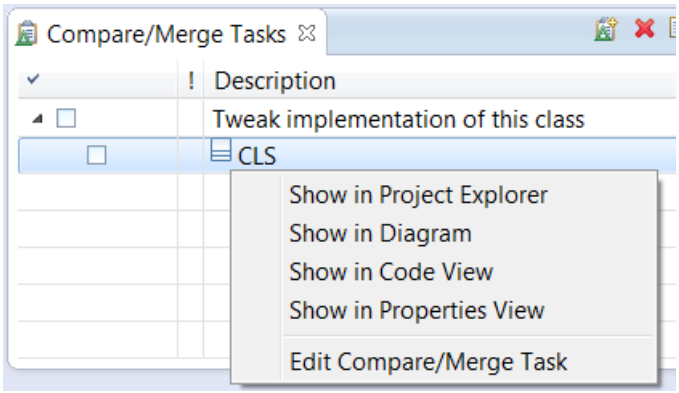
- **Compare Change Task**
  When comparing models you may notice things of interest that you want to remember. Or you may want to provide review comments on another developer's changes. If you mark a change in the change list and press the *Add Task* button () in the Compare/Merge editor

toolbar (or use the context menu), then a Compare Change task is created.



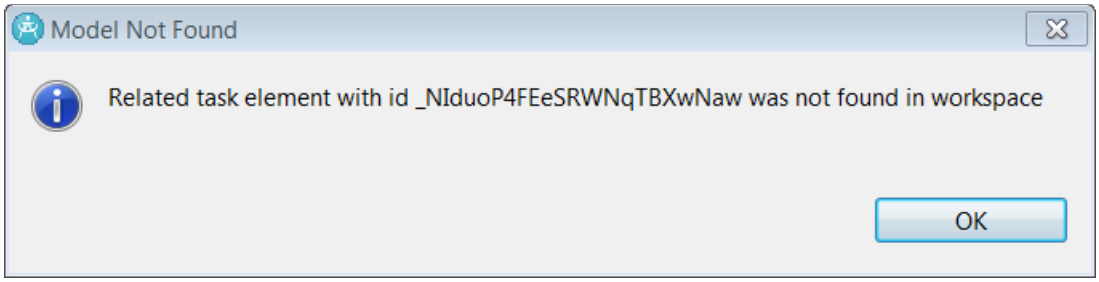
Double-click on the task to navigate to the changed element in the workspace model.

For all Compare/Merge tasks that are linked to a model element, there are some additional navigation commands available in the context menu. For example, you can navigate to the element in the Code view, in a diagram or in the Properties view.



Compare/Merge tasks are stored in your workspace metadata. To share them between different workspaces (for example between different developers) you can export them into a text file which then can be imported into another workspace. Use the *Import* and *Export* buttons that are available in the toolbar of the Compare/Merge Tasks view.

Of course, for navigation to work the workspace where you import the Compare/Merge tasks must contain the model elements that are referenced by the imported tasks. If referenced model elements are missing you will get an error message when trying to navigate to it:

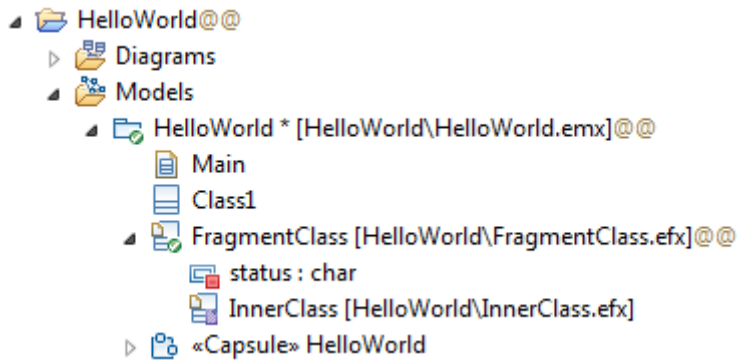


Logical Models

All but the simplest models are usually stored in multiple files. One important motivation for using multiple files, as opposed to storing the entire model in one single file, is to facilitate parallel development of the model. Ideally the model should be split into files in such a way that when developers work on different features in the model, they should not have to modify

the same files. If this is possible, the features can be developed completely in parallel without the need to merge any model files. Of course, this will not always be possible, since some features will inevitably require modifications to the same parts of the model. And in those cases you do need the ability to merge model files, which is the whole reason why the Merge tool exists in Model RealTime.

Model RealTime allows a model to be split into model files (.emx) and fragment files (.efx). Fragment files are used when a containment hierarchy has to be split on multiple files. The model file then contains the root element in the containment hierarchy, typically a package. Elements that are contained in the root element, for example classes, can be stored in separate fragment files. This continues recursively so that elements that are roots of fragment files may in turn contain other elements that are stored in separate fragment files. A set of files that store a model element and its (directly or indirectly) contained elements is called a **logical model**. The picture below illustrates this concept:



The file "HelloWorld.emx" stores the root package. It also stores some elements that are contained by this package such as the class "Class1" and the capsule "HelloWorld". However, the class "FragmentClass" is also owned by the "HelloWorld" package but is stored in its own fragment file "FragmentClass.efx". Looking inside the class "FragmentClass" we see that its attribute "status" also is stored in "FragmentClass.efx" while the nested class "InnerClass" is stored as the root of another fragment file "InnerClass.efx". In this example the logical model consists of the three files "HelloWorld.emx", "FragmentClass.efx" and "InnerClass.efx". In general, a logical model in Model RealTime consists of exactly one model file (.emx) and zero to many fragment files (.efx).

The Model RealTime Merge tool understands the concept of a logical model, and can expand the request to merge a single file, so that the entire logical model to which the file belongs gets loaded in the merge session. Thereby the Merge tool can merge all files in the logical model in one single merge session, called a **logical model merge** session. In the same way a **logical model compare** session is a single compare session where multiple files (all belonging to the same logical model) are compared. Below we primarily talk about logical models in the context of merge, but keep in mind that the logical model concept also applies when comparing models.

Logical model merge has some key benefits compared to file-by-file merge. First of all it leads to fewer merge sessions for a model that is stored in multiple files. For example, consider a model that consists of these six files:

- Model1.emx

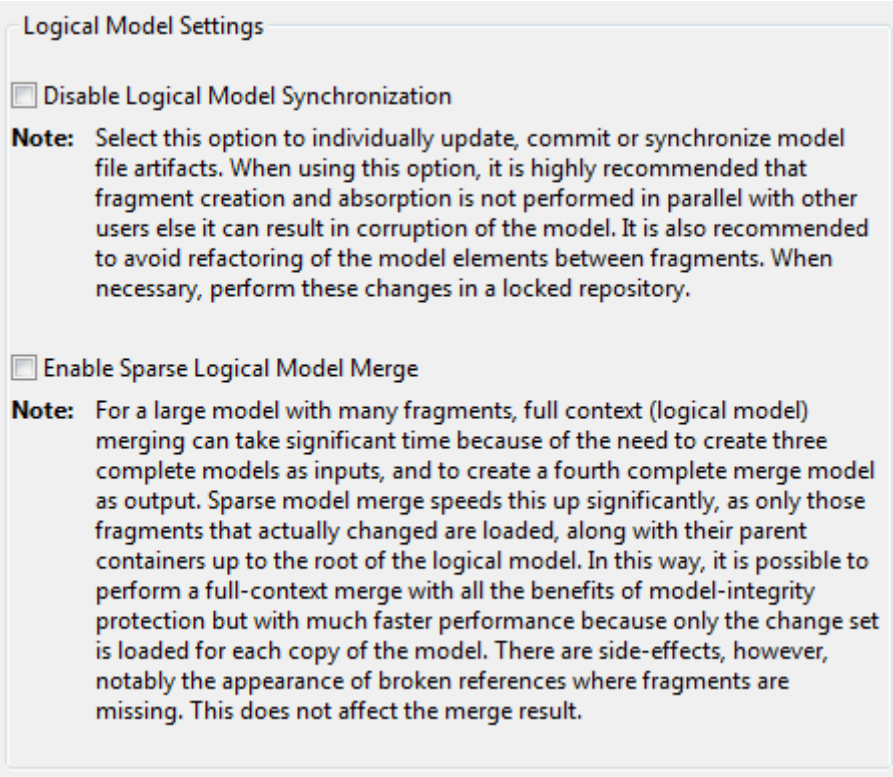
    - Fragment1a.efx
    - Fragment1b.efx
- Model2.emx
    - Fragment2a.efx
    - Fragment2b.efx

If all these six files have been modified, it would require six individual file-by-file merge sessions to merge the entire model. With logical model merge only two merge sessions are required; one for the logical model consisting of the first three files (rooted at Model1.emx), and another for the logical model consisting of the other three files (rooted at Model2.emx).

Another benefit comes from the fact that logical model merge has a bigger scope for analyzing the changes between the merged models. For example, if you would move an element from one fragment file to another fragment file within the same logical model, a logical model merge session will present that change correctly as a Move change, while with file-by-file merge you would get a Delete change when merging one of the fragment files, and an Add change when merging the other fragment file.

The most important benefit, however, is that logical model merge is better at protecting the integrity of a logical model. If you perform refactoring operations which affect the structure of a logical model (i.e. how the model is structured in model and fragment files) this leads to changes that can be really tricky to merge correctly using file-by-file merges. With logical model merge you avoid the risk for model corruption after the merge.

There is a preference *General - Compare/Patch - Modeling Compare/Merge - Logical Model Settings - Disable Logical Model Synchronization* which can be set in order to turn off the support for logical model compare/merge in Model RealTime. However, it is not recommended to set this preference if your model contains logical models that consist of more than one file. If you do set this preference, you should be careful not to modify the logical model structure on parallel development streams, since the Merge tool then will not be able to correctly merge such modifications. You should also avoid refactorings which span across different files in the logical model, such as moving an attribute from a class stored in one fragment file, to a class stored in another fragment file.



The preference *Enable Sparse Logical Model Merge* can be set if you find that your merge sessions consume too much memory and/or takes too much time to start. As explained in the note for this preference, a logical model merge implies that four, potentially quite big, models are loaded into memory during the merge session. With sparse logical model merge enabled the Merge tool will optimize this to avoid loading those fragment files that have not been modified. However, although this both speeds up the launch of the merge session, and makes it consume less memory, it is only recommended to set this preference if this is a concern for you, due to the side-effects that are mentioned in the note.

Note that not all Model RealTime integrations with CM systems support logical models, and those that do support them can do so to various degrees. Specific CM integrations can therefore have additional or different preferences related to logical model support which control the behavior in more detail. For example, the EGit integration has specific preferences in the *Team – RSx EGit Integration* preference page for enabling logical model merge and for turning on the “sparse mode” for the merge tool in order to improve performance.

Closures of Models

Above we defined a logical model as a set of files consisting of one model file (.emx) and zero to many fragment files (.efx). The reason for this definition is that Model RealTime models often are organized that way, where each logical model constitutes a kind of module which groups related model elements together into a software unit. Because the model elements in a logical model are related, most changes will only affect elements within a single logical model.

However, there will always be some changes that affect more than one model file. For example, if you move a class from one package to another (where each package is the root of their

own model file) two different logical models are affected by the change. Running a logical model merge would in this case not be sufficient as it would only look at changes in one of the logical models. Two separate logical model merge sessions would be required and they would report two independent changes (a Delete change in one logical model and an Add change in the other).

To handle this, and similar issues, we need to introduce the concept of model closures. A **closure** is defined as a set of logical models that somehow belong together. This phrasing is deliberately vague and there are several ways in which the logical models of a closure may belong together:

- They may be part of the same workspace
- They may be selected by the same working set within a workspace
- They may be selected manually by the user in the Project Explorer or some other view
- They may be the logical models that were changed as a consequence of making a group of related changes
- etc.

A closure is defined by a **closure manifest**. This is a file with the extension .ecx and it just contains a list of the model files (.emx) that together form the closure. For example, the contents of a closure file containing two models could look like this:

```
\Project1\ModelA.emx
\Project2\ModelB.emx
```

If we can obtain the source and target versions of all model files that belong to a closure, then we can merge changes in those source versions to the target versions. This is what we call a **closure merge**. You can think of it as an extension to logical model merge where we in a single merge session merge not just one logical model, but a list of logical models.

We can of course also compare changes using model closures, and such a compare session is called a **closure compare**.

A closure manifest is not something you create manually, at least not when invoking a compare or merge from the Model RealTime user interface. Rather it will be created automatically when invoking a closure compare/merge. For a closure merge, all models that have been modified, and hence may need to be merged, will be part of the closure. For a closure compare, the closure is usually taken to be all models present in the workspace, or computed from the selected files or elements.

Closure merge provides the widest possible scope for a merge session. All models that belong to the closure will be loaded in the merge session. This means that the number of merge sessions that have to be launched is significantly reduced compared to logical model merge (and even more so compared to file-by-file merge of course). Consider again the example model mentioned previously:

- Model1.emx
  - Fragment1a.efx
  - Fragment1b.efx
- Model2.emx

- Fragment2a.efx
- Fragment2b.efx

If all these six files have been modified, it will require six individual file-by-file merges to merge the entire model. With logical model merge two merge sessions are required. With closure merge only one merge session is required (the closure consists of Model1.emx and Model2.emx).

Fewer merge sessions usually means fewer merge conflicts, which makes it easier and quicker to perform the merge. The launching of a closure merge session is more time-consuming than launching a logical model merge session since the contributor and result models will be much bigger. However, if the total time is measured for all merge sessions that are required for merging a set of changes, closure merge is usually faster than both logical model merge and file-by-file merge due to the fewer number of required merge sessions (typically just one with closure merge).

A closure merge session loads the closure in four versions (left and right contributor, ancestor and result model). If the closure is big and consists of many models, this may lead to a high memory consumption for the merge session. If this becomes a concern for you, you may use the preference *Enable Sparse Logical Model Merge* mentioned above, which reduces the number of loaded files in the same way as it does for logical model merge. Specific CM integrations may have additional preferences for optimizing the performance of closure compare/merge for a particular CM system.

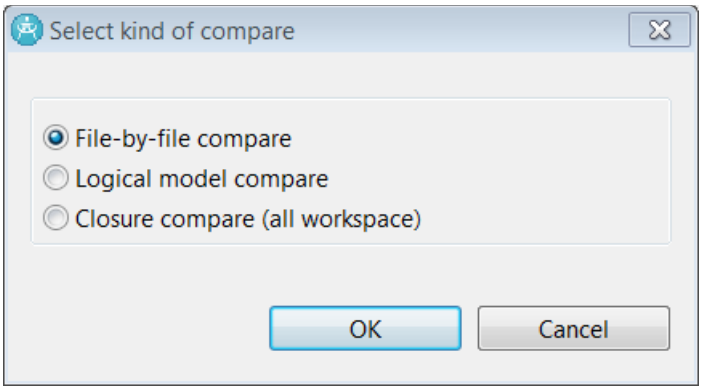
Invoking Logical Model and Closure Compare

Logical model and closure compare is currently only supported for Git (from the EGit integration user interface and from the command-line). In all other cases when you run a compare session, you will compare model files individually (i.e. a file-by-file compare).

Invocation from the User Interface (Git)

The EGit integration provides a more fine-grained control over logical models than the general on/off preference *General - Compare/Patch - Modeling Compare/Merge - Logical Model Settings - Disable Logical Model Synchronization*. With EGit you can either choose for each compare session what kind of compare to use, or you can configure it to always use a particular kind of compare. In the preference page *Team – RSx EGit Integration* you can choose between the following ways to select the kind of compare:

- **Determine compare kind from selected model element or file**
  In this case the compare kind depends on what you have selected when running a compare command. If you have selected a fragment file (.efx), or a model element stored in a fragment file, a file-by-file compare session will be launched. If you have selected a model file (.emx) or a model element stored in a model file, a logical model compare session will be launched (for the logical model of that model file).
- **Determine compare kind using a dialog**
  In this case Model RealTime will pop up a dialog when you run a compare command, asking you what kind of compare session to launch:



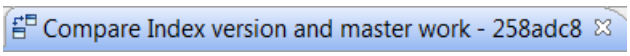
This is the default behavior, and it allows you to select the kind of compare each time you run a compare command.

- **Use file-by-file compare**
  Always launch file-by-file compare sessions.
- **Use logical model compare**
  Always launch logical model compare sessions.
- **Use closure compare**
  Always launch closure compare sessions.

Note that this preference is only used when launching a compare session based on a single selected model element or file. If you have selected multiple model elements or files, a closure compare is always launched. The closure is then a list of logical models to which the selected elements belong.

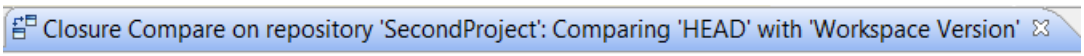
The title of the Compare editor tells you what kind of compare that was launched. Here are some examples:

File-by-file compare session:



Logical model compare session:

Closure compare session:



Invocation from the Command Line (Git)

To launch a logical model or closure compare session from the command-line (or from a script) you should use the cmcmdline.jar Command Line Tool for Compare/Merge.

The compare session will take place in an instance of Model RealTime that is running on the machine (or it can be automatically launched first). It is currently not supported to perform a non-visual logical model or closure compare. Hence the `xcompare` parameter must be used. Use the `-kind` parameter to specify the kind of compare to perform ('logical' or 'closure'), and use the `-source` parameter to specify the source version. Note that currently it's only sup-

ported to compare the source version with the current contents of the workspace of the Model RealTime instance that performs the compare operation. Therefore, if you use the `-target` parameter it will be ignored.

For example, here is the command to use to perform a logical model compare between the version tagged by “MYTAG” and the current contents of the workspace in Model RealTime:

```
java -cp cmcmdline.jar com.ibm.xtools.comparemerge.cmcmdline.CMTool xcom-
pare -kind logical -source MYTAG
```

When launching a logical model compare session exactly one logical model of the source and target versions will be compared. If the source or target versions contains multiple logical models you need to specify which one to compare using the `-filter` parameter. Otherwise you will get an error message similar to this:

```
Multiple logical models found for pattern : HEAD
Please refine your command.
```

For a closure compare multiple logical models can be compared. In this case you can use the `-filter` parameter to specify which logical models that should be part of the closure that is compared. For example:

```
java -cp cmcmdline.jar com.ibm.xtools.comparemerge.cmcmdline.CMTool  xcom-
pare -kind closure -source HEAD -filter="*Project.emx"
```

The compared closure will consist of all logical models where the model filename ends with “Project.emx”.

Invoking Logical Model and Closure Merge

Logical model merge is supported both by the ClearTeam Explorer user interface (for ClearCase) and the EGit user interface (for Git). It's also supported from the command-line (for both ClearCase and Git).

Closure merge is supported by the EGit user interface (for Git). It's also supported from the command-line (for both ClearCase and Git).

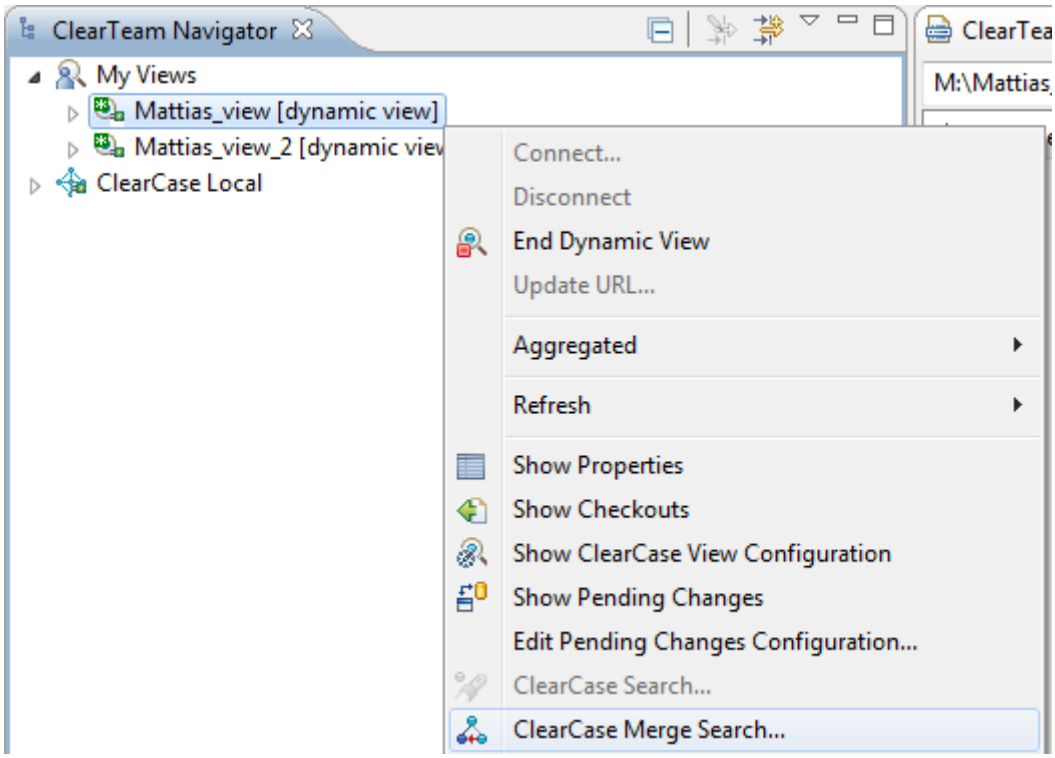
Invocation from the User Interface (ClearCase)

It is possible to initiate a merge in many different ways with ClearCase. In most of these you request a merge of a single file, and then a file-by-file merge session will be launched. This is true both when you interact with ClearCase directly, using its own user interface or command-line tools, as well as when you use the ClearTeam Explorer user interface within Model RealTime for performing the merge. Here are some examples of such situations which all launch merge of a single file:

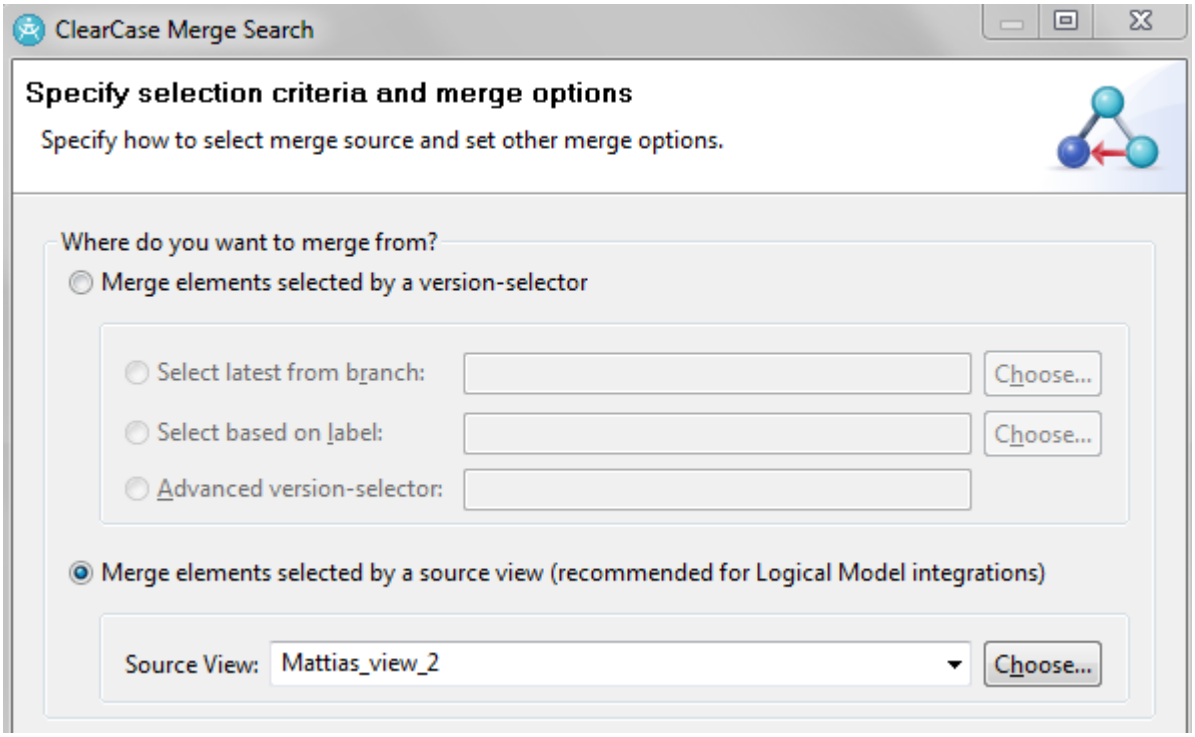
- When merging from the command-line using the command `cleartool findmerge`.
- When merging using the Version Tree Browser in the ClearCase user interface.
- When merging using the command *Merge to View Selected Version* in the ClearCase Version Tree view provided by the ClearTeam Explorer plugin within Model RealTime.

To merge an entire logical model, or several logical models, you must initiate the merge in a way that does not specify individual file versions, but rather point out a source and target context for the merge. By "context" here we mean a view, a label or a branch - concepts in ClearCase which groups related versions of multiple files together. Follow these steps to launch a logical model merge session with ClearTeam Explorer:

1. Ensure that you use a workspace that contains file versions from the target context (a ClearCase view).
2. Switch to the ClearTeam Explorer perspective.
3. Expand "My Views" in the ClearTeam Navigator and right-click on the target view.



4. Invoke the command *ClearCase Merge Search* from the context menu which launches a merge wizard. On the first wizard page just check that the target view you selected is shown as the destination view. Then proceed by clicking *Next*.
5. On the next wizard page mark "Select elements to consider for the merge" and choose the scope of the merge search, for example a particular project folder.
6. On the next wizard page mark "Merge elements selected by a source view" and choose the source view for the merge (i.e. the view that selects the file versions you want to merge from).



As seen in the dialog it is also possible to select the source files by specifying a branch or a label instead of a view. However, using a view is recommended as it is less error-prone and usually more convenient. If you use a branch you must ensure that all files have versions on that branch, and if you use a label you must ensure that all source file versions are correctly labelled.
Note that currently ClearTeam Explorer does not support to merge from source file versions that are checked out. Therefore, ensure that the files that are selected by the source view are all checked in before you proceed with the merge.

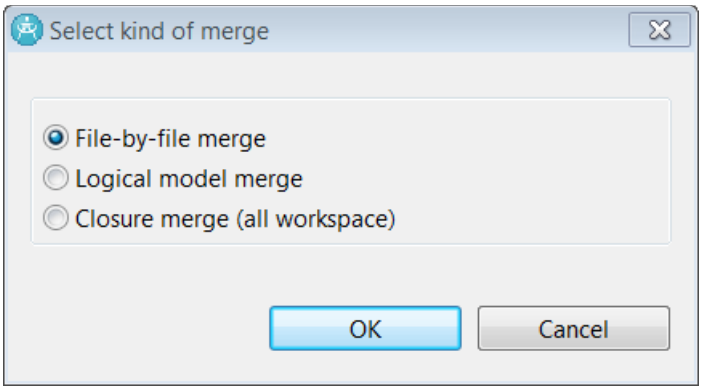
7. Set appropriate merge options at the bottom of this wizard page and then press ”Finish” to start merging.

All logical models for the files that are selected by the source view will be merged. There will be one merge session launched in Model RealTime for each logical model to be merged.

Invocation from the User Interface (Git)

Just like for compare, the EGit integration has a preference in the preference page *Team – RSx EGit Integration* for choosing between different ways to select the kind of merge:

- **Determine merge kind using a dialog**
  In this case Model RealTime will pop up a dialog when you run a merge command, asking you what kind of merge session to launch:



This is the default behavior, and it allows you to select the kind of merge each time you run a merge command.

- **Use file-by-file merge**
  Always launch file-by-file merge sessions.
- **Use logical model merge**
  Always launch logical model merge sessions.
- **Use closure merge**
  Always launch closure merge sessions.

There is a similar preference for setting the kind of merge used when launching the Merge tool. Normally you will want to use the same kind of merge both for the *Merge Model* and the *Merge Tool* commands. If you use logical model merge for *Merge Model*, then conflicts are reported per logical model. This means that all files in the logical model will be marked as conflicting regardless of in which files exactly the conflicts are located. In the same way, when using closure merge the entire closure will be marked as conflicting.

Invocation from the Command-Line (ClearCase)

To launch a logical model merge session from the command-line (or from a script) you need to use the ClearCase type manager application. It is called XtoolsTypeManager and is one of two Compare/Merge command-line tools provided by Model RealTime (see Command Line Tool for Compare/Merge).

Launch the type manager like this to perform a logical model merge (example for Windows):

```
XtoolsTypeManager.exe xmerge -logicalmerge -sourcecontext <SOURCECTX> -tar-
getcontext <TARGETCTX> -workspace <TARGETWORKSPACE>
```

Here <SOURCECTX> should be the name of a ClearCase view that selects the source versions of the files to be merged. All logical models for these files will be merged, and the versions of the files selected by the specified view will be used. If you don't want to merge versions of files selected by a view, you can instead set a label on those file versions that you would like to merge, and then use the label name as the source context. It is also allowed to use the name of a ClearCase branch as the source context, and then the latest versions of all files on that branch will be merged. However, just as is the case when launching logical model merge from the user interface, it is recommended to use a view as the source context.

<TARGETCTX> must be the name of a ClearCase view that selects the target versions of the files to be merged. Here only a view is allowed, and you cannot use the name of a branch or label for the target context. The reason is that only a view allows to specify the rules (using the config spec) needed to know how to create the new versions for the merged files.

Just like when merging from the user interface you should have the files from the target context in your Model RealTime workspace. The path to this workspace is what should be set as <TARGETWORKSPACE>. If you don't set this parameter a search will take place to find an appropriate Model RealTime instance which can host the merge session. And if no matching Model RealTime instance is found a dialog will appear to let you launch a new Model RealTime instance. However, searching for and/or launching Model RealTime takes extra time, and it is therefore recommended to specify the path to the target workspace using the `-workspace` parameter.

The initial argument `xmerge` means that all logical models will be merged visually using the merge tool in Model RealTime. This is recommended since it allows you to review all merge result models. If you instead use the argument `merge` then there will be automatic merges for all logical models without conflicts, and the merge editor will only appear for those logical models where conflicts are detected. Only use this if you are certain there are no logical conflicts in the merged models.

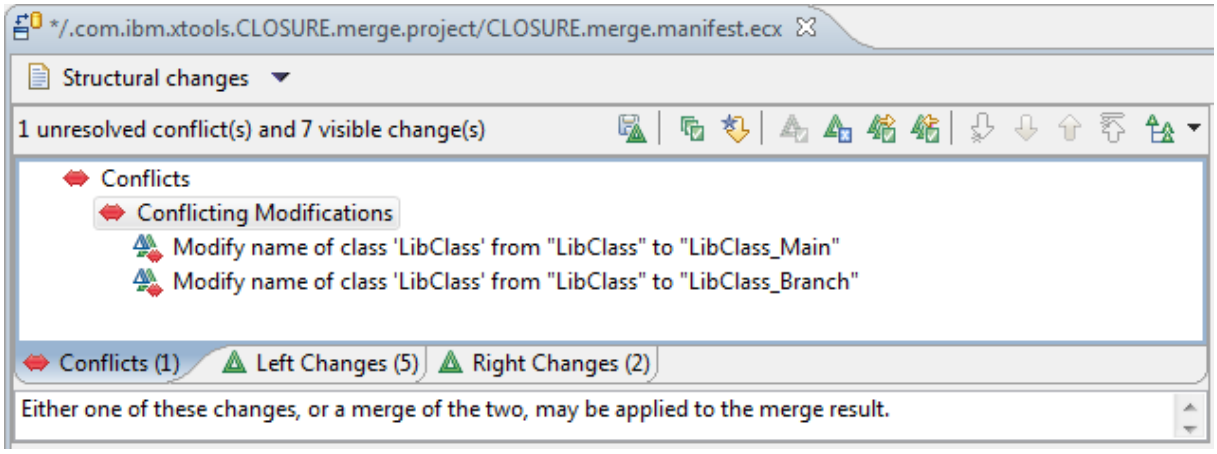
To perform a closure merge you run XtoolsTypeManager in almost the same way, except that you set the `-closuremerge` flag:

```
<XtoolsTypeManager> merge -closuremerge -sourcecontext <SOURCECTX> -target-
context <TARGETCTX> -workspace <TARGETWORKSPACE>
```

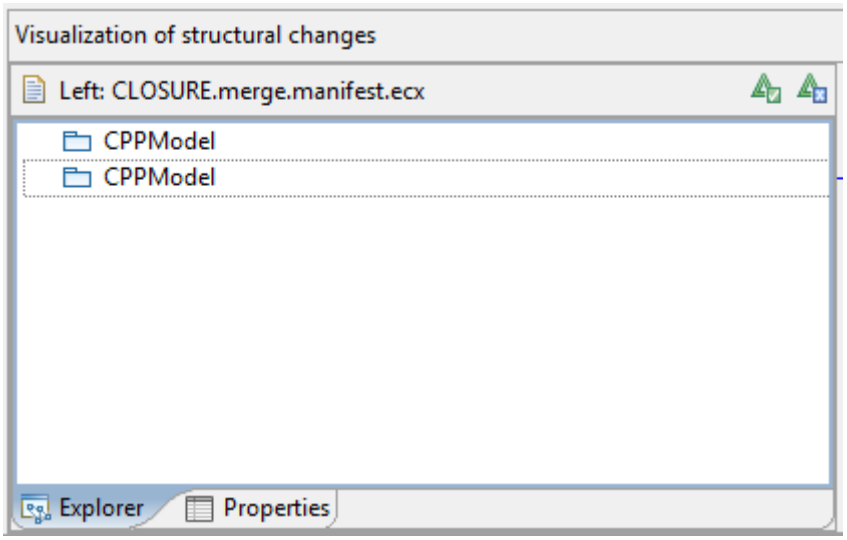
The `-workspace` parameter is mandatory when performing a closure merge, and there must be a running instance of Model RealTime with that workspace loaded.

It is recommended to run closure merge by launching the XtoolsTypeManager from a script. Such a script can automatically construct a target workspace that contains all model files that should be merged. It can then launch Model RealTime on that workspace, before finally launching the closure merge session.

The closure gets automatically generated based on the target workspace that is specified. The name of the generated closure manifest file (.ecx) shows up in the title of the merge editor in Model RealTime:



If you look at changes or conflicts in one of the Explorer tree viewers you will see that it contains each model of the closure as a root:



There are also a few other parameters that are supported by XtoolsTypeManager. Type

```
XtoolsTypeManager.exe merge --usage
```

to print detailed information about all parameters that are supported. For example, there is a parameter `-unattended` which can be used for situations when no user is available to handle merge conflicts (for example when the command is run from a nightly scheduled script). This parameter forces a non-visual merge for all logical models, and if there are conflicts these are written to a log file (specified by means of a `-logfile` parameter) which can be examined manually at a later point in time.

Invocation from the Command-Line (Git)

To launch a logical model or closure merge session from the command-line (or from a script) you should use the cmcmdline.jar Command Line Tool for Compare/Merge.

The merge session will take place in an instance of Model RealTime that is running on the machine (or it can be automatically launched first). Use the `-kind` parameter to specify the

kind of merge to perform (“logical” or “closure”). If conflicts are found files will be marked as conflicting according to the selected kind of merge. For example, for logical model merge all files in a logical model will be marked as conflicting if at least one conflict is found for the logical model. The `-source` and `-target` parameters should specify the source and target versions for the merge. The latter parameter is optional and defaults to the current contents of the workspace of the Model RealTime instance that performs the merge operation.

As an example, here is the command for performing a logical model merge from the “feature3” branch to the “master” branch:

```
java -cp cmcmdline.jar com.ibm.xtools.comparemerge.cmcmdline.CMTool merge
-kind logical -source feature3 -target master
```

If you instead use the `xmerge` parameter the Merge tool will be launched immediately after the merge has finished, if conflicts were found.

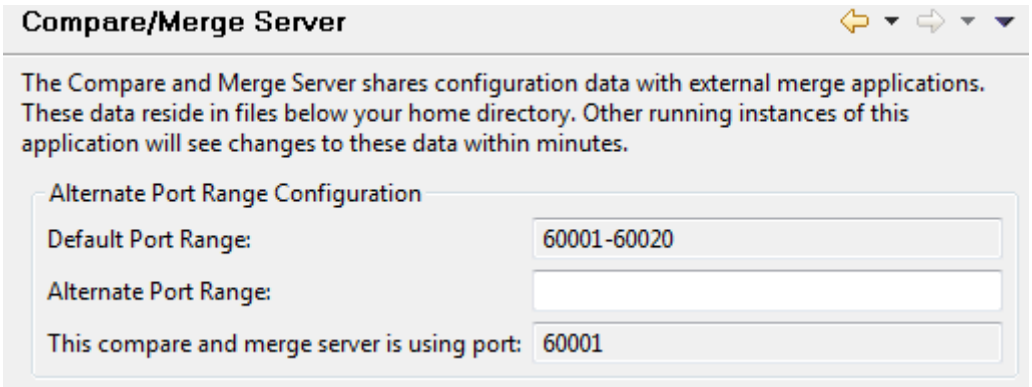
Here is an example of launching a closure merge from the version selected by the tag “pre-master” to the “feature3” branch.

```
java -cp cmcmdline.jar com.ibm.xtools.comparemerge.cmcmdline.CMTool xmerge
 -kind closure -source pre-master -target feature3
```

The closure is by default all logical models that have changes when comparing the source and target versions. You can define the closure more specifically using a closure manifest file (.ecx) and the `-manifest` parameter.

Compare/Merge Server

As we have seen above, it is possible to launch the Compare/Merge tool from outside of Model RealTime, for example from command line tools such as "cleartool". How does such command line tools find the right Model RealTime instance to use for the compare/merge session? The answer is that each running Model RealTime instance has a Compare/Merge server (sometimes also known as a Team Server) which listens on a specific port for incoming requests to perform a compare or merge operation. If you open preferences at *General - Compare/Patch - Modeling Compare/Merge - Compare/Merge Server* you can configure the Compare/Merge server. The first set of preferences control the range of ports to use for Compare/Merge servers:



By default ports in the range 60001 to 60020 are used. This means that at most 20 instances of Model RealTime can run at the same time on the machine. If you launch more than that, the additional Model RealTime instances will not get a Compare/Merge server and can hence not host compare/merge sessions that are initiated from outside of Model RealTime. You can change the port range using the preference *Alternative Port Range*. You need to do this if the default port range is not available on your machine, or if you want to support more than 20 Model RealTime instances running at the same time.

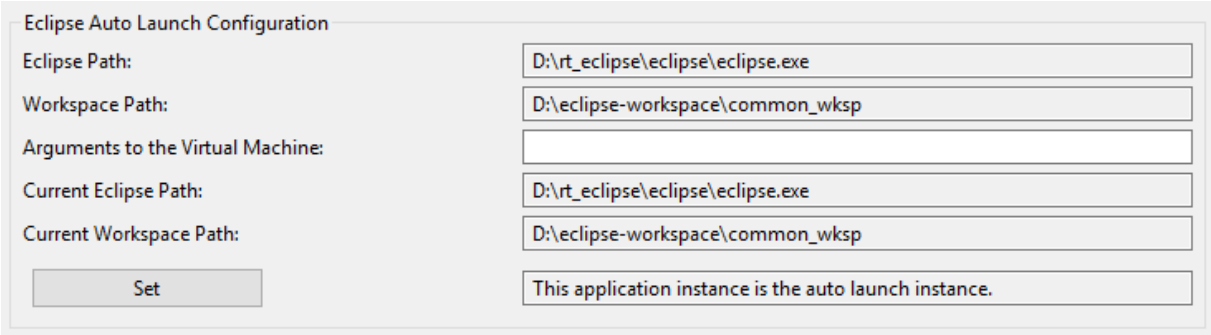
When a command-line tool (or some other tool external to Model RealTime) wants to launch a compare/merge session it will iterate over the Compare/Merge Server port range to find out which instances of Model RealTime that are running on the machine. If there are no Model RealTime instances running, a new Model RealTime instance has to be launched and the compare/merge session will be sent to it. Different command-line tools have different strategies for how to find an appropriate running Model RealTime instance, or (in case such an instance does not exist) perform an automatic launch of Model RealTime. There are in principle two strategies:

- Let the user of the command-line tool specify how to select which running Model RealTime instance to use, or how Model RealTime should be launched. The cmcmdline.jar tool (see Command Line Tool for Compare/Merge) uses this approach.
- Ask the user at run-time for this information. The XtoolsTypeManager tool (see Command Line Tool for Compare/Merge) uses this approach.

The XtoolsTypeManager will open a dialog to let you choose which of the running Model RealTime instances to use for the compare/merge session. It also let's you choose to launch a new Model RealTime instance. The Model RealTime instances are identified in the dialog by the location where they are installed as well as the workspace they are currently using. To make it easier to pick the right Model RealTime instance you can modify your `eclipse.ini` file to launch Model RealTime using the option `-showlocation`. Thereby the workspace path will be shown in the window title of Model RealTime. You can also set the preference *General - Workspace - Workspace name* to provide a description of the workspace which also will be shown in the Model RealTime window title. For example, when using ClearCase it may be useful to give the workspace the same name as the ClearCase view that is used by files contained in that workspace. These measures make it easy to pick the right Model RealTime instance for the compare/merge session.

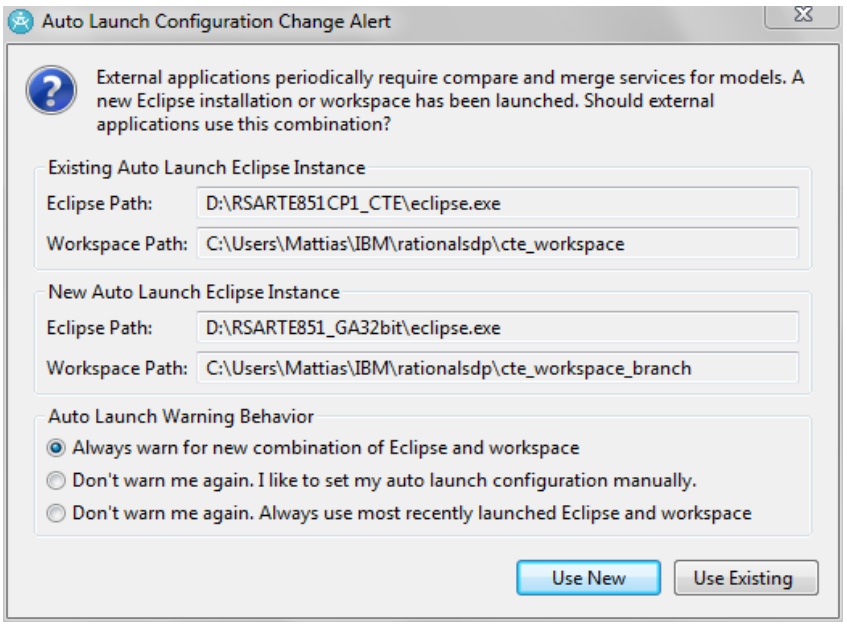
If you always want to use the same Model RealTime instance for your compare/merge sessions (or always want to launch a new instance), then you can make your choice in the dialog and then mark the checkbox *Reuse the same instance for future merges without prompting*. This setting is stored in a textfile `.PreferredRSAInstanceForMerge` that is located in the user home directory. To later make the dialog appear again you can simply delete this file.

When the XtoolsTypeManager application launches a new Model RealTime instance, the launch is done by using the second set of preferences in the Compare/Merge Server preference page which specifies the installation path, the workspace to use etc.



These settings together form something called an **Eclipse Auto Launch Configuration** and at each point in time exactly one such launch configuration is active. You can set the active launch configuration by pressing the “Set” button. The settings are stored in the file `EclipseAutoLaunchCommand.cfg` which is located in the user's home directory under `AppData/Roaming/Rational/TeamServerSharedData` (on Windows) and `rational/TeamServerSharedData/linux_x86` (on Linux).

The first time you launch an Model RealTime instance that either uses a new installation location or a new workspace, a dialog will appear to ask you whether the Eclipse Auto Launch Configuration should be set to the newly launched instance:



If you press the "Use New" button, the Eclipse Auto Launch Configuration will be updated to the new values shown in the dialog, while if you press "Use Existing" the previous settings will still apply.

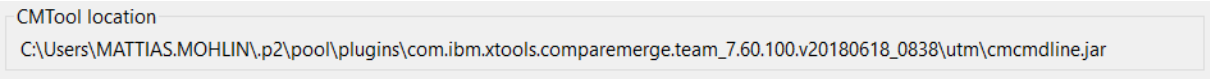
Command Line Tool for Compare/Merge

Model RealTime provides two command-line tools for launching compare and merge sessions from the command-line or scripts:

1. **XtoolsTypeManager**
   This is an executable on Windows, and a shell-script on Linux. It's located in the plugin folder for the com.ibm.xtools.comparemerge.team plugin. For example (the part in boldface depends on the installed version of Model RealTime):
   `<install-dir>\plugins\`
   `com.ibm.xtools.comparemerge.team_7.60.100.v20150603_0908``\utm`

   One way to find out the exact path to XtoolsTypeManager, which for example a script can use, is to look in the file TypeManagerLaunchCommand.cfg which is located in the user's home directory under `AppData/Roaming/Rational/TeamServerSharedData` (on Windows) and `rational/TeamServerSharedData/linux_x86` (on Linux). The path to XtoolsTypeManager is specified in this file.
2. **cmcmdline.jar**
   This is a Java application. It's located in the same place as the XtoolsTypeManager.

Hint: The full path to cmcmdline.jar can be seen in the preferences at *General – Compare/Patch – Modeling Compare/Merge – Compare/Merge Server*.



The XtoolsTypeManager is specifically designed for integration with ClearCase. It is possible to also use it as a general command-line tool for Compare/Merge, but for that we instead recommend the cmcmdline.jar Java application which is more generic, newer and more feature-rich.

This rest of this chapter describes the recommended and general command-line tool cmcmdline.jar.

Launch the command-line tool using Java like this:

```
java -cp cmcmdline.jar com.ibm.xtools.comparemerge.cmcmdline.CMTool <command> <options>
```

You should use the same Java virtual machine that you use for running Model RealTime (normally specified in eclipse.ini using the -vm argument).

`<command>` is one of the following:

- **compare**
  Perform a non-visual compare operation, i.e. a compare operation that does not launch the Model RealTime Compare user interface. The number of conflicts and changes are reported. Non-visual compare is only supported for three-way compare, so you must specify an ancestor version using the `-ancestor` option (see below).
- **merge**
  Perform a non-visual merge operation, i.e. a merge operation that does not launch the Model RealTime Merge user interface.
- **xcompare**
  Perform a visual compare operation. A connection is established to an Model RealTime instance using the Compare/Merge Server and a compare session is launched in that Model RealTime instance.

- **xmerge**
  Perform a visual merge operation. A connection is established to an Model RealTime instance using the Compare/Merge Server and a merge session is launched in that Model RealTime instance (provided that conflicts were found so that the merge could not be fully automatic).

You must also specify `<options>` according to which command that is used. The following options are available (use the `-help` option to list all available options):

- **-ancestor <file>**
  Specifies the ancestor version of a model file to use in a three-way compare or merge session. For a file-by-file compare you can omit specifying an ancestor version, and then the compare session will be two-way.
- **-autoLaunch <argument>**
  If this option is used the Compare/Merge command-line tool will not look for a running instance of Model RealTime. Instead it will launch a new Model RealTime instance and use it for the compare/merge session. The argument to this option should be the launch command to use for launching the new Model RealTime instance. Note that since this command typically contains its own arguments, you need to enclose it in double quotes to make sure it's processed correctly. For example:

  ```
  -autoLaunch "C:\rtist\eclipse\eclipse.exe -nosplash -data C:\rtist\
  egit_workspace -showlocation"
  ```

  -autoLaunchFile <file path>
  This option works the same as `-autoLaunch` but the launch command is read from a text file instead. For example, you can pass the path to the EclipseAutoLaunchCommand.cfg file in order to launch Model RealTime according to what is currently specified as the auto-launch configuration (see Compare/Merge Server for more information about the auto-launch configuration).
- **-ccpath**
  Set this option if you use any ClearCase specific paths.
- **-cformat [details | conflicts | simple]**
  Specifies how to report the result of a non-visual compare session. The default format is 'details' which means that full details about conflicts and changes are reported. For example:

  ```
  Non-Visual Compare completed with result : SUCCESS.DIFF
  conflicts       : 0
  left diffs      : 0
  right diffs     : 1
  ```

  Set the format to 'conflicts' to only show information about conflicts (second line in the example above), and to 'simple' to only show if the compare session was successfully performed and if any differences were found or not (first line in the example above).
- **-cwd <path>**
  Specifies the current working directory. Relative paths are by default interpreted as relative from the current working directory of the Java application. If they should be interpreted relative to another directory, use this option.

- **-exportSettings <file>**
  This option specifies a file to which all used option arguments are written. This also shows the default value of some options which are not explicitly set. This can be useful for throubleshooting if you get unexpected results from a compare or merge operation.
- **-fileExtension <argument>**
  As described in Switch Compare Viewer it is the file extension of the contributor files that determines which content type to use for the compare/merge session. The following file extensions are currently recognized:
  - **emx, efx**
    UML model and fragment files
  - **epx**
    UML profiles (these are recognized but currently not supported in compare/merge)
  - **bpx**
    BPMN model files
  - **topologyy, topologyv**
    Topology model and diagram files

  If your contributor files do not have these expected file extensions you need to use this option to specify which content type to use when comparing or merging these files.
- **-filter <argument>**
  Specifies a filter for a logical model or closure compare session. The filter can specify a list of model files, and only those files will then be part of the logical or closure compare session. The format of the argument is either a comma-separated list of file patterns, or `@<pattern file>` where `<pattern file>` is a file that contains such a list of file patterns. Note that the patterns are matched against the file names of .emx files. It may include the wildcard symbols `*` and `?`.
- **-help**
  Show the list of all available options, how they can be abbreviated, and a brief description of each.
- **-kind [file | logical | closure]**
  Specifies the kind of compare/merge session. The default is 'file' which means that the scope of the compare/merge session is a single file. This kind is sometimes called “file-by-file” since files are compared or merged individually, one by one. Set the option to 'logical' or 'closure' to perform a logical or closure compare/merge.
- **-lancestor <label>**
  -lleft <label>
  -lright <label>
  These options can be used to set a label for the left, right or ancestor version in a visual compare or merge session. The default labels that are used depend on the input for the compare or merge session. It can for example be a filename, or the description of a Git commit. By specifying custom labels it may be easier to understand what the contributors mean in a particular compare/merge scenario.
- **-left <file>**
  Specifies the left version of a model file to use in the compare/merge session.
- **-right <file>**
  Specifies the right version of a model file to use in the compare/merge session.

- **-log <file>**
  Redirects all log messages produced by the Compare/Merge command-line tool to a text file.
- **-manifest <file>**
  Specifies a closure manifest file (.ecx) which defines the closure for a closure merge session. If you do not specify a closure manifest the closure will by default consist of all logical models that have changes when comparing the source and target versions for the merge. The syntax of a closure manifest file is

  ```
  [closure <closure-name>]
  <path to model file>
  <path to model file>
  ...
  ```

  where `<closure-name>` is the name of the closure, and `<path to model file>` is a workspace relative path to the .emx file of a logical model.
- **-out <file>**
  Specifies the output file for a merge session. This file is sometimes called the merge result file since this is where the result of the merge is saved. This option is only used for file-by-file merges. For logical and closure merge a target workspace is used instead (see the option `-workspace`).
- **-portRange <fromPort>:<toPort>**
  Specifies the range of ports to use when communicating with the Compare/Merge Server of running Model RealTime instances. The default value is 60001:60020. You may specify a different port range if you have set the *Alternative Port Range* preference to something else.
- **-settings <file>**
  Specifies that options should be read from a text file. Each option in the file should be specified on its own line using the syntax

  ```
  <option>=<value>
  ```

  The file may contain comments by using '#' as the first character of a line.
- **-shutdown**
  If this option is set Model RealTime will be shutdown automatically when the launched compare/merge session is finished. However, this only happens if Model RealTime was also automatically launched in order to perform the compare/merge operation. If an existing Model RealTime instance is used you need to specify
  -shutdown=force
  to force that Model RealTime instance to shutdown when the compare/merge session is finished.
- **-source <argument>**
  This option needs to be set when launching a logical model or closure compare/merge session. The argument specifies the source version of the model to be compared or merged. The interpretation of the argument depends on the CM system that is used. For Git it can be one of the following:
  - The name of a branch
  - The name of a tag (including “HEAD”)
  - A commit id

- **-target <argument>**
  This option may be set when launching a logical model or closure merge session. The argument specifies the target version of the model to be merged. If this option is omitted the target version is taken to be the current workspace version of the Model RealTime instance where the merge session takes place. The interpretation of the argument depends on the CM system that is used. See the `-source` option above for valid target arguments when using Git.
  Note that for logical model or closure compare, the target version is always the current contents of the workspace in the Model RealTime instance that performs the compare operation. Hence the `-target` option is ignored in this case.
- **-threadCount <number of threads>**
  The Compare/Merge command-line tool uses threads to be able to find an available instance of Model RealTime faster when launching a compare/merge session. By default 10 threads are used. If you have many instances of Model RealTime running and you experience delays when launching a compare/merge session you can increase the number of threads used.
- **-verbose <argument>**
  This option turns on additional logging messages. It can be helpful when troubleshooting problems that seem related to how the Compare/Merge command-line tool runs. The argument should be a comma-separated list of strings that specify which modules of the command-line tool that should emit additional logging. Each string starts with '+' or '-' for enabling or disabling logging for a particular module. The following modules can be specified:
  - **PROXY** (disabled by default)
    Show details about communication with the Model RealTime Compare/Merge Server (also known as the Team Server).
  - **WSCHECK** (enabled by default)
    Show details about how the workspace pattern specified by the `-workspace` option is matched against workspace folders of running Model RealTime instances.
  - **CONNECT** (enabled by default)
    Show details about how the connection to the Model RealTime Team Server is established.
  - **EXCEPTION** (disabled by default)
    Show stack traces for Java exceptions that are thrown internally.
  - **PATH** (enabled by default)
    Show details about how paths are converted.
  - **SYSPROP** (disabled by default)
    Show details about how system properties are processed.
  - **SHUTDOWN** (disabled by default)
    Show details about the shutdown of an Model RealTime instance.
  - **RPLYSOCK** (disabled by default)
    Show details about the connection to an auto-launched Model RealTime instance.
  - **AUTO** (disabled by default)
    Show details about how an Model RealTime instance gets automatically launched.
  - **PROGRESS** (enabled by default)
    Show the progress of auto-launching Model RealTime in the console.
  - **PROGRESSSERVER** (disabled by default)
    Show progress messages in the console when using logical and closure merge.

- **ALL**
  Enables logging for all modules mentioned above.

For example, to enable all available logging:

```
-verbose=+ALL
```

You can turn off all logging by specifying:

```
-verbose=silent
```

- **-waitForLaunch <seconds to wait>**
  This option can be used when auto-launching an Model RealTime instance to use for a compare/merge session. By default the Compare/Merge command-line tool waits for at most 3 minutes for the Model RealTime instance to start up. If it has not been able to detect that it is running within that time limit it terminates. If it takes longer to launch the Model RealTime instance in your environment, you may raise this limit. The option argument is the number of seconds to wait.
- **-workspace <pattern>**
  By default the Compare/Merge command-line tool attempts to connect to the first Model RealTime instance it finds running. You may use this option to specify a pattern for the workspace folder of the Model RealTime instance that should be used for the compare/merge session. The pattern may contain wildcards ('*' and '?'). For example:

  ```
  -workspace=/home/*/myworkspace
  ```

Command-Line Integrations with CM Tools

Many CM systems can be used from the command-line. It's then convenient to have an integration with Model RealTime Compare/Merge so that compare and merge operations performed with the CM tool will launch the Model RealTime Compare/Merge tool for model files. Such integrations use the Model RealTime Compare/Merge command-line tool.

ClearCase comes with a pre-configured integration which uses the XtoolsTypeManager. That's why a command like

```
cleartool diff
```

on model files will launch an Model RealTime compare session and not the standard diff tool.

For other CM systems, such as Git, you need to configure the integration yourself. In general what is usually needed is to write a wrapper script which takes arguments from the CM system and uses some of them in order to invoke the cmcmdline.jar Java application with appropriate arguments. Note that the XtoolsTypeManager should not be used in this case, since it is specifically designed for the ClearCase integration. Refer to the CM system's documentation for the exact steps to set-up such an integration.