

Overview of the **XAM** and **XDM** models in the NetBeans IDE

Author: *Nikita Krjukov*, Saint-Petersburg Development Center

Publication Date: *July 28, 2010*

If you have any questions or comments, about this document, please send them to Nikita.Krjukov@Sun.COM

Table of Contents

Introduction.....	4
Source Code Analysis	4
XAM Module.....	4
org.netbeans.modules.xml.xam.....	4
org.netbeans.modules.xml.xam.dom.....	5
org.netbeans.modules.xml.xam.locator.....	6
org.netbeans.modules.xml.xam.spi.....	7
XDM Module.....	7
org.netbeans.modules.xml.xdm.....	7
org.netbeans.modules.xml.xdm.diff.....	7
org.netbeans.modules.xml.xdm.nodes.....	8
org.netbeans.modules.xml.xdm.visitor.....	8
org.netbeans.modules.xml.xdm.xam.....	8
How to create a new XAM-based model for a new domain?.....	8
Understanding the ModelSource Object.....	9
How Undo/Redo Works.....	10
How Transactions Work.....	12
How Synchronization Works.....	14
Automatic vs. Non-Automatic Synchronization.....	14
How XDMModel.flash() Works.....	15
How XDMModel.sync() Works.....	15
"Immutable Trees" Usage Inside the XDM Model.....	17
Calculating the Difference between XDM Trees.....	18
Merging Differences into the Main XDM Model Tree.....	18
Transmitting Modifications from XAM to XDM.....	19
Transmitting Modifications from XDM to XAM.....	19
Unique Identification of Elements in the XDM Model Tree.....	20
Domain Elements in the XAM Model.....	21
Stages of Model Transition from Valid to Invalid	21
Writing the ComponentUpdater Interface.....	23
WSDL Model: Extensibility and Support for Embedded Models.....	23

Introduction

This document explains the use and capabilities of the **XAM** and **XDM** models. It covers the most important aspects of using these models, although several smaller details are left out of scope. This document is a valuable source of information if you are going to implement your own **XAM**-based model or fix bugs in the existing models.

Both the **XAM** (Extensible Abstract Model) and **XDM** (XML Document Model) models are used to provide object-oriented access to the contents of an XML file. A separate **XAM** model should be created for each type of XML files, whereas the **XDM** model has only one implementation for all types. A connection between the XML file type and a **XAM**-based model is implemented by using the [MIMEResolver](#) mechanism and a specific `DataObject`. `DataObject` is always created first, because it provides a mechanism for constructing a specific **XAM**-based model.

NetBeans contains ready-to-use implementations of **XAM**-based models for different types of XML files, such as Schema, WSDL, XSLT, and BPEL.

Initially, an XML file can be allocated in memory only in a text form. At any moment, a request for retrieving a **XAM** model can be made. As a response to this request, an aggregate, which consists of the **XAM** and **XDM** models as well as other objects, is constructed. In this aggregate, a separate **XAM**-based model is constructed for the specified XML file type, and a new instance of the general **XDM** model that is the same for all XML file types is created.

To create a domain-specific **XAM**-based model, you need to create a set of [classes](#) that should satisfy several rules.

Source Code Analysis

This chapter describes packages that constitute the **XAM** and **XDM** models and explains the use of the most important classes. At the end of this chapter, you will better understand the structure of the underlying source code.

To know more about how certain features, such as Undo/Redo or transactions, are implemented, refer to the subsequent chapters in this document. After you familiarize yourself with both aspects, the source code structure and feature implementation, you will have a solid understanding about the entire framework.

XAM Module

[org.netbeans.modules.xml.xam](#)

This package contains the interface part of the **XAM** model. This package provides the most base interfaces and abstract classes. These classes are characterized by a high level of abstraction, therefore, they are almost not connected to **DOM**. There is only one class, `ModelAccess` that has several methods (`sync()` and `flash()`), which imply that there should be an abstract data storage. The `ModelAccess` class is a mediator between the **XAM** and **XDM** models. It represents a simplified interface to access the **XDM** model from the **XAM** model.

Another class in this package that is worth mentioning is `AbstractModelFactory`. Essentially, this class is a static centralized storage of all **XAM**-based models and **XAM** model's factories. All classes derived from the `AbstractModelFactory` class store the newly created models in the local `cachedModels` field (see the `getModel(ModelSource)` method).

The `AbstractModelFactory.getModel()` method is declared as `protected`. Unless its `protected` modifier is not overridden, all models, which have been created by an instance of the `ModelFactory` class, will be stored in the local (not static) cache of the `WeakHashMap<Object, WeakReference<M>>` type. Note that only soft references rather than direct references to the models are stored in the cache. Thus, there is no guarantee that the models stay in memory when they are held only in cache. Use the `getModels()` method to obtain a list of all cached models that were created by a specified instance of the model's factory.

The `AbstractModelFactory` class holds a list of all derived factory instances in the static `factories` field (see the class constructor). Again, rather than storing instances, this field stores only soft references to them, which is useful from memory management considerations. Using the static `getAllModels()` method, you can obtain a list of all **XAM**-based models stored in the `factories` field.

The `AbstractModelFactory` class can also broadcast events about caching new models through its listener mechanism.

This package also contains a static `SYNCER` object of the `RequestProcessor.Task` type. This static object is responsible for automatic synchronization of all XAM-based models. The need to synchronize models is checked every two seconds (the `DELAY_SYNCER` field). A model is synchronized only if it was changed and more than one second has passed since that change (the `DELAY_DIRTY` field). An unchanged model is skipped from synchronization! Thus, a delay between a change in a model and its synchronization can be 1-3 seconds.

The `AbstractModel` class contains a default implementation of the `Model` interface. In addition to ordinary functionality, it also provides a basis for supporting the following features:

•**Transactions:** `class Transaction`.

The transaction feature is embedded in this class and it is defined completely here. Transactions exist only at the XAM model layer. The XDM model does not transaction mechanism. The consistency of the XDM model is provided by its collaboration with the connected XAM-based model. The transaction feature is described in a separate [section](#).

•**Undo/Redo.**

The implementation of the Undo/Redo feature is provided mainly in the XDM module. This package provides only a small part of this feature in order to support the transaction mechanism. For more information about the Undo/Redo feature, see a separate [section](#).

The `AbstractModel` class contains two internal classes: `ModelUndoableEdit` and `ModelUnduableEditSupport`. They represent slightly modified implementations of standard classes `CompoundUndoableEdit` and `UndoableEditSupport`.

The `ModelUndoableEdit` class provides support for transaction when Undo/Redo is executed with the `redo()` and `undo()` methods. It also contains the `justUndo()` method, which is used to roll back transactions. The `addEdit()` method is also slightly modified to support `XDMModelUndoableEdit`.

The `ModelUnduableEditSupport` class redefines the `createCompoundEdit()` method in order to create `ModelUndoableEdit` instead of the standard `CompoundUndoableEdit`. This class also has the `abortUpdate()` method to roll back transactions.

•**Synchronization with the underlying model:** the `sync()` method.

* The synchronization functionality is described in this [section](#).

The `AbstractModel.sync()` method calls the `XDMAccess.sync()` method (through the `ModelAccess` interface) during a specific synchronizing transaction. As a result of this transaction, one of these state flags is set for the XAM model: `VALID`, `NOT_WELL_FORMED`, `NOT_SYNCED`. Their meanings are clear from their names.

The `XDMAccess` class is a mediator between the XAM and XDM models. Briefly speaking, the `sync()` method of this class calls the `XDMModel.sync()` method. As a result of `sync()` method execution, the entire model's aggregate gets [synchronized](#) from the bottom to the top of the chain: `XML Text --> XDM --> XAM`.

•**Keeping the model's state:** `getState()`

•**Different event listeners:** `PropertyChangeListener`

The `ComponentUpdater` interface is part of the [XDM --> XAM synchronization feature](#). This interface should be [implemented](#) in any XAM-based model.

[org.netbeans.modules.xml.xam.dom](#)

This package also contains mostly abstract classes, but at a lower level of abstraction: all of them have a connection to the underlying DOM. These classes provide interaction between the XAM and XDM models.

The `DocumentModelAccess` class is an abstract class that declares an interaction interface between the XAM model and the corresponding underlying DOM-based model. There are two non-abstract implementations of `DocumentModelAccess`: `XDMAccess` and `ReadOnlyAccess`.

The `XDMAccess` class provides a connection with the XDM model. The `ReadOnlyAccess` class provides read-only access to the underlying model by using only a standard DOM interface. `ReadOnlyAccess` is used mainly in JUnit tests.

Which class implementation—`XDMAccess` or `ReadOnlyAccess`—will be chosen at runtime depends on the

`Provider` located in the global `lookup` (see `AbstractDocumentModel.getAccessProvider()`). It is most probable that `XDMAccess` will be chosen. Its own provider is declared as a service, and if there is an `XDM` module registered in a system, this implementation will be chosen automatically.

The `SyncUnit` class is an intermediate storage of data used during synchronization between `XDM` → `XAM`. An instance of `SyncUnit` describes a set of changes related to one parent node in the DOM tree. During synchronization several such objects can be used. If a few atomic `XDM` modifications affect the same `XAM` component, such changes are accumulated in a single `SyncUnit` instance. The `SyncUnit` class provides the following data:

- `target` – a reference to the parent `XAM` component. It is specified in the constructor. It is determined by the `AbstractDocumentComponent.prepareSyncUnit` method, which recursively compares `XAM` components with `XDM` tree nodes. A [Node's unique identification feature](#) is used to perform this comparison.
- `changes` – a list of `ChangeInfo` objects
- `toRemove` – a list of nested `XAM` components that need to be removed.
- `toAdd` – a list of nested `XAM` component that need to be added..
- `removedAttributes` – a list of `XDM` attributes to be removed.
- `addedAttributes` – a list of `XDM` attributes to be added.
- `componentChanged` – a flag, which indicates the changes unrelated to either addition or removal of [domain](#) components. It usually means that one of the following lists is not empty: `removedAttributes`, `addedAttributes` or `nonDomainChanges`.
- `hasTextContentChanges` – a flag, which indicates the changes unrelated to an element or attribute. It usually implies certain objects, such as `Text`, `CData`, `Comment`
- `nonDomainedChanges` – a list of changed non-[domain](#) elements.

The `ChangeInfo` class is another temporary data storage used during synchronization between `XDM` → `XAM`. It is more finely grained than `SyncUnit`. In fact, it provides no information about a modification per se, only about its location. This storage is created for each atomic `XDM` modification, which has a certain location inside an XML document. The location is specified by a path, which is a list of DOM nodes. The path can point to either domain or non-domain elements. Depending on the element type (domain or non-domain), the content of `ChangeInfo` varies.

In case of domain elements, the entire path includes only domain elements. For non-domain elements, the `ChangeInfo` path consists of two parts: the first part includes only domain elements, whereas the second part consists of non-domain ones. In such composite paths, the first non-domain element that is located between the domain and non-domain parts is considered to be a “modified element.” If the path consists only of domain elements, the modified elements is the utmost element located at the opposite end to the root.

`ChangeInfo` contains the following data:

- `changed` – a modified DOM element determined as described above.
- `parent` – a DOM element, which is parent to the `changed` DOM element.
- `parentComponent` – a `XAM` component that corresponds to `parent`. This component is the only representative of an `XAM` model in this container. Usually it is determined at a later stage than construction of the `ChangeInfo` instance. It is cached here to prevent repetitive determination cycles.
- `domainElement` – a flag, which indicates that the path consists only of domain elements.
- `added` – shows whether the modification was an addition or deletion.
- `rootToParent` – a path from the root to `parent`.
- `otherNonDomainElementNodes` – a non-domain part of the path, which is determined as described above. If it is not empty, then `domainElement == false`.

[org.netbeans.modules.xml.xam.locator](#)

This package includes only three objects that are related to an abstract catalog.

org.netbeans.modules.xml.xam.spi

This package provides several classes used for various purposes. They include those used for validation.

Note: It looks like all unclassified classes are located in this package.

XDM Module

org.netbeans.modules.xml.xdm

This package contains only two classes: `XDMModel` and `XDMModelUndoableEdit`

The `XDMModel` class is the main class of the module. It includes many different features, such as:

- [Storing the DOM tree](#) that describes the XML file's structure
- [Synchronizing with changes](#) in the underlying Swing text document
- [Flushing the current content](#) of the XDM model into the underlying Swing text document
- Consolidation of namespace declarations
- Broadcasting events about model's modifications
- Providing an API for interaction with an XAM model through [ModelAccess](#).
- Storing the model's status
- Providing minimal threads' synchronization (synchronized methods)

The `XDMModelUndoableEdit` class is a specific instance of `UndoableEdit`. It has not been derived from [Compound](#). It describes a global change in the DOM Document. It holds two documents (or DOM trees): the old one and the new one. Both documents are part of the so called "[Immutable Trees](#)" strategy. During Undo/Redo execution, these two documents replace one another in the XDM model.

This class is special, because it can merge two successive modifications, although it does not belong to the `Compound` interface. However, such merging happens only if the new modification is also an instance of the `XDMModelUndoableEdit` class and the old DOM document of the new modification is equal to the new DOM document of the old modification. In practice, this scenario occurs in case of two successive modifications, when the new document of the new modification simply becomes the new document of the old modification. The new modification itself and the previous new document of the old modification are lost.

org.netbeans.modules.xml.xdm.diff

This package contains classes, which are related to *a)* the [algorithm of determining differences between two XDM models](#) and *b)* also to the [algorithm of merging differences into the main tree](#).

The `NodeInfo` class provides temporary information about a single DOM node during constructing a list of differences. It includes the following:

- `node` – a reference to a DOM tree node.
- `pos` — a position number inside the parent element.
- `ancestors1` — a chain of DOM nodes, which is a path from the `node` to the root of the old DOM tree. The first item in this chain is the `node`'s parent. The last item is the root of the DOM tree. The `getOriginalAncestors()` method returns this list.
- `parent2` — a new parent of a DOM tree node. It is assumed that due to the "[Immutable Trees](#)" mechanism, a tree node can be part of multiple trees. Therefore, it might have many parents. When a node is modified, it can acquire a new parent.
- `ancestors2` — a chain of DOM nodes, which is a path from the `node` to the root of a new DOM tree. It's the same as `ancestors1`, but is related to a new DOM tree. The new root appears as a result of tree mutation. The first item in the chain is `parent2`.

The `NodeInfo.NodeType` enumeration contains four types of node: `ELEMENT`, `ATTRIBUTE`, `TEXT`, `WHITE_SPACE`

Despite the fact that `NodeType` is declared inside `NodeInfo`, it is not used there. `NodeType` can be obtained from `Node` by using this method:

```
public static NodeInfo.NodeType getNodeType(final Node child)
```

The `Difference` class represents an atomic difference between two XML documents. It includes `NodeInfo.NodeType` and two interrelated instances of `NodeInfo`. These two `NodeInfo` instances share the same `ancestor1` and `ancestor2`, as well as `NodeType`.

The `Difference` class is an abstract class, which has three direct descendant classes: `Change`, `Add`, `Delete`. They represent three specific modification actions.

Among these three descendant classes, only `Change` is non-trivial, so we will describe it in more detail below.

The `Change.Type` enumeration describes the following modification types: `TOKEN`, `ATTRIBUTE`, `POSITION`, `UNKNOWN`, `NO_CHANGE`. (Note that the two last types are not used anywhere in the source code, and have been added here just in case).

It is not mandatory for a modification to be assigned to only one of these types, as they are used more like flags.

The `Change` class is derived from `Difference`. Its purpose is to provide detailed information about the modification.

In the `Change` class constructor, there is a non-empty list of `Change.Type` objects. This list is formed by the `DiffFinder.checkChanges(Node p1, Node p2)` method. The order of items in this list is not important. The purpose of this list is to show that a modification can have a set of attributes.

Another important functionality of the `Change` class is to determine the differences in attributes. Using the methods outside of this class, one can only confirm the fact of attribute modifications (`DiffFinder.checkAttributesEqual()`). The `Change` class (more precisely, its constructor) determines what attributes were modified. The result of this determination is a list of the `Change.AttributeDiff` objects. The `Change` class has three child classes: `Change.AttributeAdd`, `Change.AttributeDelete`, and `Change.AttributeChange`.

org.netbeans.modules.xml.xdm.nodes

This package contains classes that describe the XDM tree. All these classes are implementations of DOM interfaces and are derived from the base `NodeImpl` class. This base class introduces the notion of the [unique identifier \(ID\)](#) and provides a connection to the XDM model (`getModel()`). In addition, it provides basic functionality needed to work with the DOM tree.

This package also contains several classes related to XML parsing: `XMLSyntaxParser`, `Token`, `TokenType`.

The `Convertors` class is a utility class.

org.netbeans.modules.xml.xdm.visitor

This package contains a definition of the base visitor—`DefaultVisitor`—for the XDM model and a set of derived visitors, which implement specific tasks. They are used in different places. The meaning of each visitor is clear from its name.

org.netbeans.modules.xml.xdm.xam

This package contains three classes, which are responsible for XDM <--> XAM interactions. They are described in more detail in the sections about [synchronization](#).

How to create a new XAM-based model for a new domain?

NetBeans provides a set of existing ready-to-use XAM-based models for several XML file types, such as Schema, WSDL, XSLT, and BPEL.

If you want to create a new XAM-based model, perform the steps below:

- Create a new class derived from `AbstractDocumentModel`. This class will correspond to the new model.
- Create a base XAM component for the new model. It should be derived from `AbstractDocumentComponent`.
- Create a set of components derived from the base class. These should correspond to the structure of the XML file. (Note: It is typical for many models that declaration and implementation of each component should be done separately. For example, declaration can be implemented as Java interfaces. In this case, one can at first create a hierarchy of interfaces, and at a later point, a hierarchy of implementations. However, we do not insist on implementing this approach).
- Create a Factory class to construct all new model components. Usually, you have a separate method for each component and a general method for creating components by names.
- Implement the `ComponentUpdater` interface as described [below](#).
- Register `QName` of each component in the [domain's name set](#).
- If you plan to use your new XAM-based model in combination with a UI editor, additional efforts might be needed to ensure proper integration with the XAM-based model and a correct work of the Undo/Redo feature. For more information, see the description of [Undo/Redo](#).
- Create a `ModelSource` object for the model. Be aware about its [hidden vulnerabilities](#).
- Create a new Visitor interface to support iterations over the tree of model components. As it is necessary for most implementations, so it is worth creating it from the beginning.
- More instruction can be found here <http://xml.netbeans.org/xam-usage.html> (Note: Please be aware that this instruction has not been updated for some time, so we are not responsible for its contents).

Understanding the ModelSource Object

`ModelSource` keeps all the information that is necessary for creating a model and its identification. It is used as an argument for the `AbstractModelFactory.getModel(ModelSource)` method. If the model was created earlier and it is still available in cache, you can take it from cache. If the model is not found in cache, a new model is created and added to cache. The `ModelSource` object is used as a key to search in cache and also to create a new instance of the model.

`ModelSource` is not a key by itself. The key object is constructed using the `AbstractModelFactory.getKey(ModelSource)` method. The default assumption is that the key is an object of `DataObject` or `Document` type (see class `...xml.xdm.xam.XDMAccessProvider`). However, the way of key construction can be redefined. You can find an example of using `FileObject` as a key in JUnit tests (see `...xml.schema.model.resolver.FileObjectModelAccessProvider`).

The functionality for creating a new model is not completely defined at the level of an abstract XAM-based model. However, it is known that to create a new model, a `Document` object should be provided by `ModelSource`. This document is used as the source of an XML text.

All the above-mentioned data is indirectly by `ModelSource` inside of the nested `Lookup` object. On the one hand, this organization allows us to keep any data there. On the other hand, this makes it difficult to understand the source code, because there is no easy way to know what is held inside the `Lookup` object. Unfortunately, it is even difficult to understand this using the debugger, because `Lookup` often has an inconvenient implementation of the `toString()` method. So, a good practice becomes to thoroughly document what should be inside of the `Lookup` object for each XAM-based model implementation.

The `ModelSource` object also contains the `isEditable()` flag. This flag enables you to create read-only models. However, there is an architectural error that comes to light in certain cases. For example, if a model is already created and cached once, then the cached `isEditable` flag will always be returned, even if it is different from the real one. This cached `isEditable` value will remain until NetBeans is reloaded.

There is another vulnerable moment related to the `ModelSource` use. An XAM-based model holds a direct reference to the instance of `ModelSource` that was used during model's creation. After being created once, the cached instance of

the model is usually returned upon subsequent requests. Each time the model is requested, different instances of the `ModelSource` object are returned. Therefore, only at the first request, the internal `ModelSource` is equal to the `ModelSource`, which was used during the request. Although these two instances of `ModelSource` usually have the same `DataObject` and `FileObject`, but there is a possibility that they hold different `Document` objects.

Using the model's cache is inevitable, because it greatly improves performance. But it is necessary for the model to be in strict correspondence with the source XML text (see `AbstractDocumentModel.getBaseDocument()`). In practice, two different `Document` objects can be set after the UI editor was closed and reopened again. The root cause is that a new instance of `Document` is created each time the UI editor opens. To enable correct synchronization between the `Document` object and the model, a workaround approach needs to be used (as in `...xml.retriever.catalog.Utilities.createModelSource()`).

The problem with cached `ModelSource` was the reason for [issue #166414](#).

How Undo/Redo Works

The question of how the Undo/Redo feature is implemented stands aside from the purpose of this paper. However, it is quite important for general understanding, because this feature noticeably influenced the architecture of `XAM` and `XDM` models. Let's take a look at the most important aspects of this feature:

- **Creating `UndoableEdit`.** The Undo/Redo functionality is based on using `javax.swing.undo.UndoableEdit`. The main methods of this interface are `undo()` and `redo()`. Any user modifications are transformed to objects, which implement this interface. It allows one to roll back these modifications and apply them again.

- **Collecting changes.** All modifications are collected in the form of a two-sided chain. By using Undo/Redo, the user can step over this chain of modifications in both directions. The base functionality of Undo/Redo is located in the `javax.swing.undo.UndoManager` class. A chain is created separately for each edited file. NetBeans has a quite sophisticated editor functionality. An editor can consist of several views. For example, the Schema Editor has three views. An editor can be cloned. All editors share one storage, which is the `ClonableEditorSupport$CESUndoRedoManager` class to keep Undo/Redo chains..

- **Applying and rolling back changes.** When the user invokes the Undo/Redo commands, they initiate applying or rolling back the corresponding changes in the chain of modifications. A special position counter implemented in the `UndoManager` class points to the current position in the chain. It is important to note that the direction of moving along the modification chain is controlled by `UndoManager`, and it is not meaningful for the Undo/Redo commands.

A NetBeans-based XML editor usually includes two views: XML source editor and a special graphical view. In addition, the Navigator view and Property Sheet are also used. With this in mind, we distinguish two types of modifications:

- Text modifications, which occur when the user changes the XML text in the source editor.
- `XDM` model modifications. They occur as a result of calling the `XAM` model API methods from the graphical editor, property sheet, or Navigator.

Text modifications are considered external from the viewpoint of the `XAM/XDM` models. These modifications are out of scope for this document. The second type of modifications is generated inside the `XDM` model and represented by the `XDMModelUndoableEdit` class.

According to the architecture of a multi-tab editor, each tab can have its own `UndoManager`. For example, the `MultiViewPeer.privateGetUndoRedo()` method requests an active `MultiViewElement` to return `UndoManager` with `getUndoRedo()`. The returned object is of the `UndoRedo` type.

Usually all tabs of an XML editor share the same `UndoManager-xml.xam.ui.undo.QuietUndoManager`. This is a `XAM`-related wrapper and replacement for `ClonableEditorSupport$CESUndoRedoManager`, a standard Netbeans `UndoManager`. For example, take a look at how the `SchemaEditorSupport.getUndoManager()` method is used. Note that Undo/Redo management is performed by the "XAM Common UI" module, not by the `XDM` model. The `QuietUndoManager` class is a good starting point for learning the Undo/Redo functionality. You can put breakpoints at its `undo()` and `redo()` methods in order to debug the Undo/Redo commands.

The `QuietUndoManager` class provides a slightly different approach to the [standard way](#) of handling the chain of

modifications. There is a significant difference between modifications initiated from text and from XAM API. Text modifications are more finely grained: addition or removal of only a single character in the text can be treated as one atomic modification. Such fine-grained modifications should be ignored by the graphical XML editor. Thus, a new kind of a modification chain needs to be introduced. This chain is more complex in comparison with the standard one. All modifications made in the text mode are grouped together. If the user switches to the graphical editor and invokes Undo, the entire group of modifications is rolled back. But if the user switches back to the source editor and invokes Redo, these modifications are rolled back one by one, not as a group.

This feature was intentionally implemented to hide intermediate states from users when they are working in the graphical editor. As intermediate states can often be invalid, showing them in the graphical view might cause numerous error messages indicating that the model is in invalid state. Therefore, getting rid of the special `UndoManager` might have resulted in an inadequate Undo/Redo behavior in the graphical editor.

To enable correct work of `QuietUndoManager`, it is necessary to maintain and switch between different editing modes, depending on the current editor tab. This is implemented by using the `setModel(xam.Model)` method. When the current view is the source editor, the model is set to `null`. Otherwise, it is set to the current XAM model.

Let's return to discussing collections of modifications. The `QuietUndoManager` class inherits listening to events within `UndoableEditEvent` from the base `UndoManager` class. These events are used to pass modifications from the place where they have been generated to the place where they are collected. Owing to the listening capabilities, `QuietUndoManager` can collect modifications.

When the XML source editor is active, the text document is listened for modifications. When another editor is active, the XAM model is listened. You can see this mechanism in the `SchemaEditorSupport.addUndoManagerToModel()` method.

Note that when the user works in the XML source editor, only text modifications are being collected. The `XAM --> XDM --> XAM` synchronization happens in parallel, independently of the Undo/Redo mechanism. Objects of the `XDMModelUndoableEdit` type can be generated during synchronization, but they are lost because `QuietUndoManager` does not listen to the modifications of the XAM model in source editing mode.

When the user switches from the source editor mode to another editor, listening to the XAM model is activated and listening to the text model is deactivated. In this case, modifications are initiated only through the XAM API.

Let's take a closer look at how XAM modifications are handled. At first, the `XAM --> XDM` synchronization occurs, during which a new root of the XDM tree and a new `XDMModelUndoableEdit` object are created. The new `XDMModelUndoableEdit` is sent to the XAM model (because it listens to `UndoableEditListener`). The XAM model adds `UndoableEdit` into `CompoundEdit`. Later, after the modification transaction finishes, the `XDMModel.flash()` method is called to actually modify the XML Text. This text modification leads to generation of new `UndoableEditEvent` events, which are accumulated inside a new `CompoundEdit` object. Then these events are sent through the listeners mechanism to the XAM model as well. As a result, the `UndoableEditSupport` object of the XAM model now contains a special instance of `AbstractModel$ModelUndoableEdit` that includes two `UndoableEdit`. The first instance of `UndoableEdit` is related to the XDM model's modification, and the second one is related to the XML text's modification. These instances are independent from each other, and in case of the Undo/Redo command they are applied to their respective models separately, one by one. For an example of this mechanism, see the `CompoundEdit.undo()` method, which is called from `AbstractModel$ModelUndoableEdit.undo()`. This approach has a certain redundancy as it stores both the XDM and XML text modifications. However, it allows us to avoid calling the `flash()` method after Undo/Redo, because this might be time consuming, especially in case of large XML documents.

As the result, the chain of Undo/Redo modifications can contain alternating blocks of text and XDM modifications. If a text editor is active, then movements over the chain goes according the order. When a graphical editor is active, XDM modifications are processed according to the queue, but a series of text modifications is handled at a single step.

Now let's return to discussing how modifications are applied and rolled back. When the user invokes the Undo/Redo commands, `UndoRedo` is requested from the current editor and the `undo()` or `redo()` methods are called. In our case, `QuietUndoManager` is returned as an instance of `UndoRedo`.

If you want to take a deeper look at how Undo/Redo is implemented, please note that despite the "XAM Common UI" module is documented better than XAM/XDM, the source code for the Undo/Redo feature is quite complex. For example, it is common to see that `UndoableEdit` objects are used to wrap one another. To change the way of how a command is executed, the object can be wrapped with another command that has additional features. For example, upon the first textual change, the following chain of wrapped `UndoableEdit` objects is created:

- `ClonableEditorSupport$BeforeModificationEdit`
 - `CompoundUndoManager$BeginCompoundEdit`
 - `BaseDocument$AtomicCompoundEdit`

The deepest object is generated by the source editor and the upper ones are added to enable additional actions.

Additional aspects:

- In the `QuietUndoManager` class, `Model.sync()` synchronization occurs at the end of executing the `undo()` and `redo()` methods. Theoretically, this additional synchronization is redundant, because everything should be synchronized without it. However, this was obviously implemented in order to ensure that the `XDM` to `XAM` model are synchronized in any case.
- Owing to the "Immutable Trees" approach, the `XDM` model can include multiple trees, or, more precisely, multiple roots of trees. However, the `XDM` model is not capable of storing any references to these multiple roots. Actually, only current root is referenced by the `XDM` model. All other roots can be held in memory only due to the references from the `XDMModelUndoableEdit` objects, which themselves are kept by Undo/Redo changes' list inside of the `ClonableEditorSupport$CESUndoRedoManager` class.

Questions:

- How to initiate Undo/Redo through an API? It is necessary to get access to the `UndoManager` class, which collects the chain of modifications, and then call the `undo()/redo()` method. As the Undo/Redo mechanism is mostly external regarding the `XAM/XDM` models, in most cases the `UndoManager` class needs to be requested from the `UI editor` rather than from the model. However, the `AbstractModel` can be used to perform an [Undo to roll back transactions](#). If Undo is required just for rollback purposes, this should be done in the same manner as it is implemented with `AbstractModel`.

In practice, calling Undo/Redo through an API is used in JUnit tests. For testing purposes, a new instance of `UndoManager` can be created. You can find numerous examples in the JUnit tests of the `XDM` model.

How Transactions Work

The transaction management mechanism is implemented in the base implementation of the `XAM` model. Note that this mechanism can be overridden in descendant models, such as the BPEL model. In this section, we describe how transactions work in the `XAM` model.

The primary class responsible for transaction is `AbstractModel`. It has several methods related to transactions: `startTransaction()`, `endTransaction()`, `isInTransaction()`, and `rollbackTransaction()`. The `XAM` model allows only one transaction at a time. Parallel and nested transactions are not supported. An attempt to start another transaction from a parallel thread results in a thread lock until the current thread finishes its transaction. An attempt to start another transaction by a thread, whose previous transaction is still running (**nested transaction**), throws an `IllegalStateException`. Presently, there is no differentiation into read and write transactions. An attempt to modify a model outside of a transaction again results in a `IllegalStateException`. However, reading a model outside of a transaction is allowed. In fact, reading a model outside of a transaction is error prone, because this does not prevent another thread from starting a transaction and modifying the model. So, reading a model inside a transaction looks safer.

Let's take a closer look at how to call `XAM` transactions. Previously, the following template for calling transactions was recommended:

```
model.startTransaction()
try {
    doSomethingWithModel();
    model.endTransaction();
} catch (Exception ex) {
    model.rollbackTransaction();
    processException(); // rethrow or process exception if nessasary
}
```

There was a requirement (before issue [181290](#) is fixed) that either `endTransaction()` or `rollbackTransaction()` can be used to finish a transaction. Otherwise an `IllegalStateException`

appeared. But it turned out that this restriction was error prone, so it was eliminated. Now, the new template is recommended for use:

```
model.startTransaction();
try {
    doSomethingWithModel();
    //
    // the transaction will end here if there wasn't any exception.
    model.endTransaction();
} finally {
    //
    // the transaction will be rolled back here if it wasn't ended.
    model.rollbackTransaction();
}
```

Be aware that the new template works well only after the mentioned fix was integrated (not included in NetBeans 6.9!). Another one important conclusion is that it's not recommended to use the `model.isInTransaction()` method inside of transaction templates, because it provides information about the model, but not about the transaction. So it returns true if the previous transaction completed and another one can be started by another thread.

Also, it is better to avoid any manipulations that support nested transactions. The model itself does not support nested transactions. Therefore, any attempts to support them will result in vulnerable solutions. You can find additional explanations in [my comments to issue #181290](#).

Note that transactions happen not only during XAM model's modifications, but also during Undo/Redo and model synchronization. In total, there are three sorts of transactions:

- Ordinary transactions, initiated by the XAM model's implementation code
- Synchronization transactions
- Undo/Redo transactions

All these sorts of transactions have slightly different execution algorithms. The `AbstractModel` class has special indicators to track the difference: `inSync` and `inUndoRedo`. All kinds of transactions are executed one by one without overlapping. Therefore, changes in an XML text cannot be processed at the same time with any other modifications made through the XAM API or those initiated by the Undo action.

The transactions mechanism is related with the Undo/Redo feature. Multiple atomic modifications can be done during a single transaction; but eventually only one `XDMModelUndoableEdit` modification is added to the Undo/Redo chain. This modification includes the XMD model states before and after the transaction. This is why all modifications made within a single transaction are treated by Undo/Redo as a single step.

Rolling back transactions is implemented as a separate complex process. The Undo functionality is utilized to implement rollbacks. This might look quite suspicious, because an XAM model can exist without any `UndoManager`. As we discussed above, the Undo/Redo functionality is external and is managed from a `Multiview` graphical editor. If there is no such an editor, there is no `UndoManager`, modifications are not collected at all, and thus there is no component to which the `undo()` method can be applied. However, such considerations are incorrect, because the XAM model has its own simplified Undo/Redo implementation. The model contains the `UndoableEditSupport` class, whose `beginUpdate()` method is called when the transaction starts, and the `endUpdate()` method is called when the transaction ends. Due to such approach, all atomic `UndoableEdit` modifications made during a transaction are collected inside `UndoableEditSupport.compoundEdit`. In case the transaction need to be rolled back, the `CompoundEdit.undo()` method is called. However, note that unlike in case of a regular Undo/Redo, a new transaction will not be started for this Undo mechanism, because it is already executed inside of another pre-existing transaction.

We have not found many source code examples of rolling back transactions. This mechanism is mainly used in JUnit tests in the Schema module. This might mean that this feature was added later, after most of the XAM-based models had been written.

Possibly it might be a good idea to improve the transaction mechanism as follows:

- To separate read and write transactions, similarly as it was done in the BPEL model. Later, leverage this code in all XAM based models. Such separation is important mainly from the performance viewpoint and it is more convenient.

- To use `Runnable` and `Callable` (or a similar mechanism) instead of the transaction template described above. Such approach is now used in the BPEL model.

- Roll back transactions automatically in case of any exception. Now if an exception is thrown, a transaction finishes in an intermediate state, which can be incorrect, and the entire model might be invalid. This feature is most important for the BPEL model, because rolling back is not implemented there at all.

How Synchronization Works

Synchronization is a process of bringing two or more dependent data storages into a consistent state. In our case, we have the following chain of storages:

- XML File
- Swing text Document
- XDM Model
- XAM Model

This chain is created automatically during XAM model initialization. As it was mentioned [above](#), the XAM and XDM models form an aggregate. In fact, an XDM model is wrapped by an XAM model.

Modifications made at any level of the chain are should be propagated to all levels up and down. Here are the most typical cases:

- 1.Modifications of the XML text in the embedded source editor. The Swing Document is changed first.
- 2.Modifications of an XAM model within a transaction. It is usually initiated by user actions.
- 3.Modifications initiated by Undo/Redo
- 4.Modifications of the XDM Model as a result of a process such as refactoring or auto-formatting.
- 5.Modifications of XML File in an external text editor.

If the user makes changes in the Navigator window or another non-text editor, these changes are made in the XAM model through its API and then they are propagated downstream along the XAM --> XDM --> XML Text chain.

If the user introduces changes in an embedded source editor, first these changes are made in the text, and then they are propagated upstream along the XML Text --> XDM --> XAM chain.

In case the user activates Undo, these changes are recorded in the XDM model first. Then they are propagated in both directions along the XML Text <-- XDM --> XAM chain .

Modifications are propagated from XAM to XDM (XAM --> XDM) by using the `DocumentModelAccess`: `setAttribute()`, `removeAttribute()`, `appendChild()` methods. See the detailed description of these methods in the [separate section](#).

Modifications are propagated from XDM to XAM (XDM --> XAM) by using the event broadcasting mechanism of the XDM model and the `ComponentUpdater` interface on the XAM model's side. See the detailed description of this process in a [separate section](#).

Modifications are propagated from XML Text to XDM (XML Text --> XDM) through the `XDMModel.sync()` method. It transforms the internal content of the XDM model in accordance with the underlying Swing text document.

Modifications are propagated from XDM to XML (XDM --> XML) through the `XDMModel.flash()` method. In this case, the text document is partially or completely overwritten.

Automatic vs. Non-Automatic Synchronization

In a new instance of an XAM model, automatic synchronization is enabled by default. The synchronization is performed [asynchronously](#). Using the `AbstractModel.setAutoSyncActive()` method, you can either enable or disable automatic synchronization. If automatic synchronization is disabled, a custom external code is required to maintain explicit synchronization between the XML Text and XDM/XAM models.

The `Model.sync()` method should be called in order to explicitly start synchronization (as in case of using `QuietUndoManager.syncModel()`). Note that as the `AbstractModel.sync()` method starts a new

transaction, it must be called outside of another transaction.

In most cases, there is no need to disable automatic synchronization. If explicit synchronization is required for any reason, it can be instantly initiated through the `sync()` method, regardless of whether automatic synchronization is on or off.

How XDMModel.flash() Works

The `flash()` method transforms the content of the Swing text document in accordance with the XDM model. The XDM model implicitly references the text document through `ModelSource`. It is implied that there is an instance of the `org.netbeans.editor.BaseDocument` class in `lookup`. For more details, see [this section](#).

The `flash()` method performs the following work:

- It verifies that the model is in a stable state. There is no sense in calling the `flash()` method when the state is unstable; `IllegalStateException` is thrown otherwise.
- A new instance of the `FlashVisitor` class generates the text of a new document. This Visitor class implements the `XMLNodeVisitor` interface through the following chain:
`XMLNodeVisitor --> DefaultVisitor --> ChildVisitor --> FlashVisitor`

The purpose of the visitor is to iterate over all tree nodes and serialize everything to a text. As XDM tree nodes keep all text tokens including different white spaces, the initial view and formatting of the original text document are preserved.

- A new local instances of `UndoableEditListener` is temporarily created and attached to the text document. It will listen to any modifications made in the document.
- Through the `Utils.replaceDocument()` method, the new content is added to an existing document. A special optimization approach is used:
 - The old and new versions of the text are compared.
 - The old text is split into three areas. It is implied that the first and last areas are the same in both documents, whereas all differences are located in the middle area.
 - The middle area is copied from the new text to the old one.
- After the new content is added, the listener mentioned above receives an `UndoableEdit` message. This message is further resent from behalf of the XDM model, and the temporary listener is removed. [Here](#) you can find an explanation what the Undo/Redo event is used for.

The `flash()` method is called in two cases:

- When using the `prettyPrintXML()` method

The `prettyPrintXML()` method performs minimal auto-formatting of the text. It eliminates line brakes, indents and so on.

- At the end of an XAM transaction (`AbstractModel.endTransaction()`). The `flash()` method is called from the `XDMAccess` class. It happens at the very end of [transaction processing](#). If the transaction is related to synchronization, the `flash()` method is not called because of the backward data movement from the XML Text to the XDM and XAM models (`XML Text --> XDM -> XAM`).

How XDMModel.sync() Works

The synchronization process is divided into two logical phases (two methods): `prepareSync()` and `finishSync()`. The first phase can be initiated in advance, for example, by using the `ModelAccess` class in the XAM module. If the preparation has not been done in advance, it starts automatically by the `sync()` method. Apparently, this division is made for the sake of thread performance optimization. In fact, the `AbstractModel.runAutoSync()` method is the only place in the source code where the phases are executed separately and in different instances of `RequestProcessor.Task`.

The following steps are performed at the first (preparation) phase:

- The current XDM model's status is saved in order to restore it at the end of the phase. The status can be `BROKEN`,

`STABLE`, `UNPARSED`, or `PARSING`. Then, the status is set to `PARSING` until the end of the preparation phase.

- An absolutely new `XDM` tree is constructed by the current content of the Swing text document. The construction itself proceeds in two stages. The first one is performed by the `XMLLexer` class, the second one, by the `XMLSyntaxParser` class. .

- The Language property is set to the Swing document in order to force the usage of `XMLLexer`.

- A new Document is constructed by using the `XMLSyntaxParser` class. The Document is the root item of the tree. The Parser receives a sequence of tokens generated by `XMLLexer`.

- The current `XDM` tree is requested by the `getCurrentDocument()` method. The current tree corresponds to the latest stable (valid) state of the `XDM` model. The method returns `null` if it is loaded for the first time.

- Then, the so-called `preparation` object is created as the result of the first phase. This object is temporary and auxiliary. It is used to pass data between the two phases. It has the fixed `SyncPreparation` class. This object contains the following data:

- The root of the old tree (`oldDoc`)

- The root of the new tree (`newDoc`)

- The list of differences between the two trees. The items in this list are derived from the `Difference` class.

- An optional instance of `IOException`, which indicates a fault case. The exception wraps another exceptions, which can happen during the preparation phase.

The `preparation` object can be created in different ways:

- If an exception happens during the preparation phase, it is wrapped into `IOException` and put into the `preparation` object. Other fields are empty.

- If the current tree has not been initiated yet (`== null`), the `preparation` object has only the new tree root (`newDoc`). Other fields remain empty again.

- If the current tree is already initiated, more complex actions are executed. A new instance of `XDMTreeDiff` is created. It builds the list of differences between the old and new trees (for details, see a separate [section](#)). Then, a new `preparation` object is created, which contains the old tree (`oldDoc`) and the list of differences. Note that eventually the new tree turns out to be redundant, because all changes are added to the old tree by using the “[Immutable Trees](#)“ mechanism.

- Finally, the initial `XDM` model's status is restored with an intention that it will be updated during the second phase.

The second phase is performed by the `finishSync()` method in the following order:

- It is checked that the preparation phase has completed. Otherwise, the phase is interrupted.

- If the `preparation` object contains `IOException`, the `XDM` model is assigned the `BROKEN` status and the exception is thrown again here. It interrupts the sync phase.

- The initial model status is saved just in the same way it was done during preparation. The model is set to the `PARSING` state.

- The current tree root is obtained again with the help of the `getCurrentDocument()` method.

- If the model was initiated for the first time (`preparation.newDoc != null`), the new root is simply considered current. Also, the tree components are recursively connected to the `XDM` model with the `addedToTree(XDMModel)` method.

- Otherwise a merging process starts, which is merging differences into the current model's tree. The merge is executed by the `MergeDiff.merge(List<Difference>)` method. For details, see [this section](#). Before merging, the current tree root is checked for being the same as it was during the preparation phase.

- Notification events about the `XDM` model's changes are distributed by the `fireDiffEvents(List<Difference>)` method. It takes the same list of differences as used for merging. All events have the same standard type `PropertyChangeEvent`. The subscribers have the `PropertyChangeListener` type. There are only 3 kind of Property: added, modified, deleted. The values of

properties have `NodeInfo` type, which is an auxiliary object. It contains the changed tree node itself and detailed information about its location in the tree. You can find more details in separate [section](#).

These events are intended for [XDM --> XAM synchronization](#). In fact, the only subscriber is the `XDMListener` class.

- If the current tree root is different from the one obtained at the beginning of the second phase, this means that the tree was modified. In this case, a new `UndoableEditEvent` event is initiated and an instance of the `XDMModelUndoableEditEvent` class is used.
- The `XDM` model's synchronization is considered successful if no exception was thrown. The model's state is set to `STABLE`. Otherwise, the state is set to `BROKEN` and a new `IOException` is thrown.
- Finally (in the final block), the current tree root is rolled back to the previous one if the model is unstable (`!= STABLE`). This measure guarantees that in case of an error, the `XDM` model will correspond to the latest stable state!
- The `preparation` object is set to null after the synchronization finishes. This indicates that the next synchronization can be started.

"Immutable Trees" Usage Inside the XDM Model

"Immutable trees" is a special approach to trees mutation. It is very important for the `XDM` model. Being a special implementation of the `DOM` interface, the `XDM` model represents the hierarchical structure of an XML document. A traditional `DOM` model is mainly oriented to reading an XML document, whereas the `XDM` model supports XML document modifications.

All items of a `DOM` tree can be separated to leafs and nodes. Leafs have a value but do not have children. An XML attribute is a good example of a leaf item. Nodes do not have associated values, but can have children. Let's suppose a primitive, degenerate case when a tree consists of the two items: a root node and a nested leaf. If it is necessary to add another leaf item (inside the node), it looks like the node needs to be changed because it should contain two children instead of one. There are two approaches, which can be applied in such a case: mutable and immutable trees. In case of mutable trees, it is implied that the node contains a list of children and a new leaf is added to the list. In case of immutable trees, a new instance of a node is created, which contains two nested leafs instead of one. Note that the node is the root item. Because the new root node was created, it is considered to be the root of the tree.

Now it is not quite clear what happens with immutable trees in a more general case. Imagine that we have a big tree and the same modification happens somewhere deep inside it: a new leaf item is added to a parent node. But now the parent node is not the root and it has its own parent node. That parent node is also considered modified because one of its children has been modified and recreated. Taking into account this assumption, we eventually come to the tree root. As a result, any atomic mutation results in the creation of a new chain of items, which links the changed item with the tree root. The rest remains unchanged. Creation of a new modified node is copying all children from an old node with the exception of only one child in the list. Creation of a new modified leaf is construction of a new item with a new value.

Both approaches have their own pros and cons:

- Mutable trees
 - During mutations all items remain parts of a single tree. In this case, children can explicitly reference to their parents.
 - If it is necessary to keep several versions of the same tree, these versions are in fact full copies of the trees.
- Immutable trees
 - Upon each mutation, a new root node is created. However, the previous one remains unchanged, and children became shared by both the old and new trees. So a child cannot reference its parent. Such trees can be unidirectional only.
 - Keeping several versions of the same tree almost does not require additional efforts. Moreover, it is very compact by nature. It is very important for the Undo/Redo feature.

For more information, see the "**Second approach: immutable trees**" section in [Tree data structures](#).

Calculating the Difference between XDM Trees

Before reading this section, it might be useful to read about [auxiliary classes](#).

Two XDM trees are used to find a difference. One of the main tasks is to find corresponding items in the trees before comparing them, because it makes little sense to compare irrelevant items. Two special methods—`DiffFinder.findMatch()` and `findMatch()`—are used to match items.

`DiffFinder.findMatch()` looks for similar child items inside the same parent.

If a child item is `dom.Element`, the comparison is done by using a special `ElementIdentity (XAM)` service object. Note a chain of derived classes `ElementIdentity --> DefaultElementIdentity --> XDMUtil.XDElementIdentity`. The latest class is used by default, although it can be overridden. The comparison of nodes is made by the "id", "name", and "ref" key attributes. By using `ElementIdentity`, you can detect cases when the order or children changed. In this case, the modification is more compact. Otherwise, the two types of modifications (deletion and addition) will be generated. The identification algorithm for different XAM-based models can slightly vary (see the `AbstractDocumentModel.setIdentifyingAttributes()` class) in additional identification attributes.

The `findMatch()` method compares to items of the `dom.Text` type.

The `DiffFinder.ChildInfo` class collects intermediate information about children of a tree node. The `DiffFinder.findMatch()` method discovers matching children pairs, which are collected inside `DiffFinder.ChildInfo.compareNodeMap`.

The `DiffFinder.ChildInfo.siblingBeforeMap` field maps an element to its previous sibling. It looks like it was done for optimization.

The `DiffFinder.ChildInfo.posMap` field maps a child to an array with two integer values (the array is of size 2). Both integer values are order indexes. The first integer is the absolute order number of the child item. The second integer is also an order number, but among another children of the same type. For example, for a child item of the `Text` type, the array can be [7, 3] which means that the item is the seventh in an absolute order, but it is the third `Text` child of the parent. The `DiffFinder.ChildInfo.getIndex()` method returns the first index. The `DiffFinder.ChildInfo.getPosition()` method returns the second index.

The `DiffFinder.SiblingInfo` class calculates a child item in another way.

Eventually, building of the list of differences is performed by the `DiffFinder.findDiff()` method. It takes two DOM documents and compares them recursively with the help of the `compareChildren()` method. A list of differences is the result of this comparison.

Another method—`findOptimized()`—is called later to optimize the list of differences. During optimization, some differences related to reordering children are eliminated from the list. A new `Map` object is filled in before optimization (`deMap`). It has a parent node as a key and the list of relevant differences as a value. So the differences relevant to the same parent node are collected together in the map. Then differences that are relevant to the parent node are processed one by one.

The `DiffFinder` class has two specific descendants: `NodeIdDiffFinder` and `XDMUtil.XDUDiffFinder`.

The `NodeIdDiffFinder` class is used from the `XDMModel.resetDocument()` method. It implements a simplified algorithm of node identification by node IDs only. These IDs are supported by the XDM model internally (see details [here](#)). The class has the redefined `DiffFinder.findMatch()` method. It is implied that both of the tree nodes compared are `xdm.Node`, so they both have unique identifiers which can be obtained through the `getId()` method.

The `XDMUtil.XDUDiffFinder` class has a slightly different algorithm for comparing XML attributes. It is used only within the `XDMUtil.compareXML()` method which itself is used only for JUnit tests.

Merging Differences into the Main XDM Model Tree

During the `XML Text --> XDM synchronization`, the new text is parsed and an absolutely new XDM document tree is created. After this, the new tree is compared to the main model tree and a list of differences is calculated, [as described above](#). This section explains how the differences are applied to the main tree. They are applied according to the “[Immutable Trees](#)” concept and eventually the main tree gets a new root. Merging starts from the `MergeDiff` class

and its only one public `merge()` method. The method is called during the second stage of synchronization (see the `finishSync()` method). The main purpose of this class is to sort out the list of differences into separate tree nodes. So all differences related to the same node are collected together. Then the differences are applied to the nodes one by one. Applying differences is delegated to the `XDMModel` class.

As described in the "Immutable Trees" section, a new chain of nodes appears in the `XDM` tree after mutation. It starts from the root and goes till the modified item. The chain is generated by the `XDMModel.mutate()` and `XDMModel.updateAncestors()` methods. The first method applies a mutation to the modified item and then the second one updates parent nodes recursively.

The `XDMModel.mutate()` method performs general and standard actions for different types of model mutations. The specific part of mutations is implemented separately and is accessible through the `XDMModel.Updater` interface.

Transmitting Modifications from XAM to XDM

Both the `XAM` and `XDM` models hold similar data. But the main part of data is located in the `XDM` model.

Let's take a look at the general structure of the `XAM` model. The `XAM` components form a tree just like the `XDM` components do. All components are derived from the `Component` interface. When a new `XAM`-based model is created there usually is the following class inheritance:

- `Component` – the base interface for all `XAM` components.
- `AbstractComponent` – the first base implementation. At this level, there is no linkage to an `XDM` model. The tree structure is introduced at this level. An instance of an abstract component references the parent component and stores a list of its children. It also references the `XAM` model.
- `AbstractDocumentComponent` – another base class. At this level, there is a linkage to the `XDM` model. Each component has a direct reference to an `XDM` model component. There is one-to-one correspondence between the `XAM` and `XDM` components.
- `AbcComponent` – the base class implemented in the ABC model (here, ABC is a pattern). This class is base for all the components of the ABC model. For example, the `SchemaComponent` is a base class for all components of `SchemaModel`.

Note that the `XAM` model holds only information about XML Elements. There is no information about attributes, text, etc. Any `XAM` component corresponds to an XML Element. The information about attributes is held only inside the `XDM` model. When an element is added or removed, changes are added to both the `XAM` and `XDM` models. When an attribute is added, removed or modified, the changes are added only to the `XDM` model.

Any changes to a `XAM` component or its attributes are added to the corresponding `XDM` model directly by using the `DocumentModelAccess` class methods. As it is mentioned in the "Immutable Trees" section, any mutation of the `XDM` model leads to creation of a new chain of nodes that forms a new tree. A direct reference from a `XAM` component to an `XDM` component needs to be updated if it points to an `XDM` component, which was replaced during mutation. The purpose of the `DocumentModelAccess.NodeUpdater` interface is to do this. The chain of new `XDM` nodes is transferred back to the `XAM` model through the `DocumentModelAccess.NodeUpdater.updateReference` method. During this, references to the modified `XDM` nodes are recursively updated in the corresponding `XAM` components (see the `AbstractDocumentComponent.updateReference()` method). A special mechanism for `XDM identification` is efficiently used here.

Any modification of the `XAM` model (through the API) can be done only inside a transaction. The transaction can contain numerous atomic mutations like adding an attribute or changing the attribute value. Mutations are distributed in the `XAM` and `XDM` models straight away. However, they appear in the XML Text only at the end of the transaction. The `XDMAccess.flash()` method is used to move the changes to a text. If a transaction is rolled back, all mutations inside the `XAM` and `XDM` models are also rolled back. You can find explanation of this process in [this section](#).

Transmitting Modifications from XDM to XAM

Transmitting modifications from `XDM` to `XAM` starts from broadcasting events by using `PropertyChangeEvent`. But the `XAM` model does not listen to these events. They are listened by the `XDMListener` class. This class is a part of

an aggregated set of classes, which are constructed during initialization of the XAM model (XAM model + XDMModel + XDMAccess + XDMListener). XDMListener does not permanently listen to the events from the XDM model. The subscription is activated by the startSync() method and is deactivated by the endSync() method. In fact, the methods specify the time frame for the XDM --> XAM synchronization process. This process can be activated in two cases: a) when Undo/Redo is executed and b) when synchronization is executed as a result of changes in an XML document.

After the XDMListener is activated, it starts collecting mutations. Each mutation is stored as a SyncUnit object. The collected SyncUnit objects are applied to the XAM model right after the synchronization finishes (endSync()). So the XAM model remains unchanged until the very last stage of the synchronization process.

The mutations are applied by using the AbstractDocumentModel.processSyncUnit(SyncUnit) method. If an error happens during processing, an Exception is thrown (it must not be an IOException), which interrupts the synchronization. Then the XAM model should be automatically regenerated with the AbstractDocumentModel.refresh() method, which is called from the finally section inside the AbstractModel.sync() method. Regeneration of a XAM model is more time consuming than applying small changes. So it is used only in emergency cases.

As mentioned above, the XAM model holds information only about XML elements. Nothing is changed in the XAM model structure if an attribute or text is changed in the XDM model. In this case, the XAM model only broadcasts PropertyChangedEvent events. If a mutation affects an XML element, its processing is delegated to a special object, which has to implement the ComponentUpdater interface. Generally speaking, there are only two operations: ADD and REMOVE. Both are described inside ComponentUpdater.Operation. The CHANGE case is does not make sense because an XML element itself does not have a state. It only has children. But its state is held by its attributes or text items, which are independent of both XAM and XDM models.

The XAM module does not have the default implementation of the ComponentUpdater interface. Therefore, each XAM-based model should its own implementation of this interface. Such an Updater implementation is responsible for addition or deletion of various model components and also for broadcasting the PropertyChangedEvent events. For details about implementing ComponentUpdater, see [this section](#).

Unique Identification of Elements in the XDM Model Tree

An XDM tree is much similar to a DOM tree. The XDM tree items implement DOM interfaces. However, they have additional properties, such as, for example, a unique identifier – ID, which is an integer number.

This ID has a special lifecycle supported by the XDM model. It is assigned to an item only once and remains unmodified until the item is deleted. Any mutations of the item do not result in the ID change, even though a new item is created (in accordance with the “[Immutable Trees](#)” concept). After mutation, a new item is created and it receives the same ID, which its predecessor had.

A new ID is assigned only to absolutely new items. A simple counter is used to determine new IDs. Theoretically, there might be a counter overflow, but the probability is very low.

Note that sometimes mutations are treated as a complete removal of an item and a subsequent addition of a new item, though this cannot be visible from the user point of view. For example, when a targetNamespace attribute of a root node is modified, the previous content of the document turns out to belong to another namespace. This case is treated as deletion of all the content and addition of the new one. Therefore, the IDs are recalculated in this situation.

The identification feature is vital for [XDM --> XAM synchronization](#). At some moment, it is possible that the XDM model has already been modified, but the XAM model has not yet been modified. Each XAM element has a direct reference to the corresponding XDM element. But at this moment the reference points to the XDM element from the old XDM tree. After synchronization, all references have to be switched to the corresponding XDM elements in a new XDM tree. However, because both the old and new elements can have the same IDs, there is a possibility to find the XAM component and update its reference. Below you can find several helpful methods that handle identification:

- `NodeImp.isEquivalentNode(Node)` checks if the nodes have the same ID.
- `AbstractDocumentModel.findComponent(...)` looks for an XAM element that is relevant to the the XDM element.
- `AbstractDocumentComponent.updateReference(...)` refreshes the reference of the XAM element to the XDM element.

Domain Elements in the XAM Model

Elements of any XML file can be divided into two groups: meaningful and insignificant. For example, documentation tags can be treated as insignificant. Usually, a special class that is derived from `Component` is created inside the XAM model for each meaningful XML element. Therefore, the XAM model knows and works only with meaningful elements. A set of meaningful elements comprises a domain, and these elements are called “domain elements.”

The `AbstractDocumentModel.getQNames()` method returns the set of names (`Set<QName>`) of domain elements. This set is specific for each XAM-based model implementation. For example, it is different for the Schema and WSDL models. Eventually the set is used to filter out modifications which are relevant to a specific XAM based model. It is necessary for separate processing of domain relevant modifications during the [XDM --> XAM synchronization](#).

Note that non domain elements are not only those elements, which are not enumerated inside of a set, but also other DOM tree nodes like attributes, text or `CData`.

During a new XAM based model development it is important to add new elements to the domain set after adding a corresponding class to a model.

Stages of Model Transition from Valid to Invalid

Both the XAM and XDM models have the `state` field, which indicates the current state of the model. However, the meaning of the “state” is different. In general, the state of a XAM model indicates whether the model is valid. The validity needs to be confirmed between transactions. The state of the XDM model represents stages of synchronization. It is meaningful only inside a transaction and makes little sense between them.

This section explains different states of XAM models and cases when these states change. In general, there are two main state transitions: from valid to invalid and vice versa.

When a model is in a valid state, it means that all the XML Text, XDM and XAM levels are synchronized with each other. Now let's imagine that a mutation occurs. As mentioned above, mutations can be initiated in different ways: from the source text, from the XAM model or by the Undo/Redo commands. A first glance, it might seem that the third way (Undo/Redo) is just a degenerate case of the first or second ones, as Undo/Redo are in fact only postponed changes initiated by either source text changes or XAM model changes. However, from the implementation viewpoint, the third Undo/Redo case is worth describing as a separate case of mutations.

1.If a mutation is initiated from a XAM model, it happens inside a transaction. The transaction can either be applied or rolled back as a whole. The actions similar to Undo happen in case of a rollback. The XDM model root pointer is switched back to the previous tree root (see the “[Immutable Trees](#)“ section) and then the XDM --> XAM synchronization is initiated. It leads to the XDM model first and then the XAM model comes to the general state, which preceded the transaction. As a result of the synchronization, the validation state of a XAM model is redetermined. It must be the same as it was before the transaction.

If the transaction is successfully committed, then the XAM --> XDM synchronization is not executed at all and the validation state of a XAM model is not updated. It goes in such a way because the changes happen directly inside the XAM and XDM models just during the transaction. The only action that happens after the transaction finishes is flashing changes to XML Text and broadcasting events to the XAM model subscribers.

There is an important consideration here. It is impossible to implement any mutations through the XAM model API if the model is in the invalid state. It happens so because changes are to be done inside a transaction (see the `AbstractModel.validateWrite()` method). In turn, a transaction cannot be started if the model is invalid (see the `AbstractModel.startTransaction()` method). The conclusion is that regardless of whether a transaction was successful or not, the XAM model **always remains in a valid state**.

2.If a mutation is initiated from Undo/Redo, processing depends on whether the initial mutation was made through the XAM API or through an XML Text. It is very important here to differentiate the notion of an initial mutation and the mutation induced by Undo/Redo. An initial mutation is performed by user and it results in the creation of a new Undo/Redo mutation. An Undo/Redo mutation is often called `UndorableEdit`. After creation, it is added to a special chain. For details, see [this section](#).

If the initial mutation was made in the text, the associate Undo/Redo mutation is processed by the text editor first. Then the XDM model is notified through event subscription and it happens absolutely like the mutation made by a user manually.

If the initial mutation was made through the XAM API, the associated Undo/Redo mutation is an instance of the `XDMModelUndoableEdit` class. It references the previous XDM model tree root. When such a mutation is applied, the XDM model is switched to the old tree root (with the help of the `XDMModel.resetDocument()` method). Then the XDM --> XAM synchronization is initiated and eventually the validation status of the XAM model is updated. However, [as mentioned above](#), such mutations always switch the XAM model from a valid state to another valid state, so in fact the validation state does not change.

Note that such an Undo/Redo mutation does not lead to the XDM --> XML Text synchronization (flashing), because the same `UndoableEdit` [contains](#) the text related to `UndoableEdit` inside, which is applied to the text independently. Apparently, this was made for performance improvements at the expense of memory usage and general reliability.

Another feature of these Undo/Redo mutations is that they are executed inside specific pseudo-transactions (see `AbstractModel$ModelUndoableEdit.redo()`). In case of `Exception` during Undo/Redo processing, the XAM model tree is fully recreated (see the `AbstractDocumentModel.refresh()` and `AbstractDocumentModel.createRootComponent()` methods). The new XAM tree is regenerated based on the existing XDM tree. Under normal conditions, the XAM tree is created only once and is updated during XDM --> XAM synchronizations.

3. If a mutation is initiated from XML Text, the end-to-end XML Text --> XDM --> XAM synchronization occurs. If any exceptions or rule violations occur during this synchronization, the XAM model can be considered invalid. To summarize, this is the only reason that can result in an invalid state of the XAM model.

Important: after transition from a valid to an invalid state, the internal structures of the XDM and XAM models **remain in the state corresponding to the last valid state**. This feature is part of the initial design, and it should be kept during further model improvements. In case of a failure, the XDM model switches its tree root back to the previous one. The XAM model can receive and implement actual modifications only after the XDM model has been successfully updated. You can find a more detailed description in [this section](#).

Even when synchronization fails, it is critical to have both the XDM and XAM models **always synchronized**, because the synchronization mechanism is based on propagating changes. It is a great advantage from the performance point of view, but it makes synchronization vulnerable in general. If at some point the XDM and XAM models are out of sync, subsequent modifications cannot return them to the synchronized state. A complete rebuild of the XAM model is required after that.

There is a special case when the XML Text --> XDM synchronization is successful, but an error happens during the XDM --> XAM synchronization. The XAM model needs to get the temporary `NOT_SYNCED` validation status and its content has to be rebuilt. Rebuilding involves creation of a new root component of the XAM model. It is generated based on the root XDM component. The `AbstractDocumentModel.refresh()` method is responsible for it. If everything goes right, the XAM model state is discarded back to `VALID`. It is important that the `NOT_SYNCED` status can remain only in case an error happens during the `refresh()` method execution and it indicates a serious damage of the XAM model.

The details of the described mechanism can be found here:

```
org.netbeans.modules.xml.xam.NsPrefixCreationTest.testInterruptedComponentUpdater()
```

Theoretically, this rule should guarantee that the XAM and XDM models are always in sync. Unfortunately, in practice the models can sometimes lose synchronization. For example, an important part of synchronization is the `ComponentUpdater` interface, which is responsible for the addition and deletion of components in the XAM model. The interface should be implemented for each XAM based model. An incorrect implementation can lead to losing synchronization and eventually the XAM model becomes inconsistent, though it has the `VALID` status. It is quite difficult to detect the moment when a model becomes inconsistent because it is not always visible from the UI. But some time later, this might cause various exceptions. For more information, see [this section](#).

Let's take a look at how a model is returned **back to a valid state**. Following the same logic, we can conclude that the only way to return back is to undergo the same XML Text --> XDM --> XAM synchronization. Such synchronization can be initiated either by manual text modifications or as the result of Undo/Redo. The current XAM model validation state is not important to synchronization at all. It is reassigned to the XAM model after the synchronization is finished. Thus, the process is completely the same as in case [#3](#).

Writing the ComponentUpdater Interface

As mentioned above, `ComponentUpdater` is an interface, which needs to be implemented by each `XAM`-based model. It executes the last stage of the `XDM --> XAM` synchronization. It instructs the `XAM` model to add or delete components. The main part of processing is done before the `Updater` is initiated. For example, in case of adding a new component, the component itself is already constructed by the `DocumentModel.createComponent()` method.

Implementation of the `ComponentUpdater` interface can be complex or trivial, depending on the requirements of a specific model. In a primitive case, the only thing that needs to be done is adding a child component to a parent and vice versa. There are two methods to do it: `AbstractComponent.insertAtIndex()` and `AbstractComponent.removeChild()`. An example of such a primitive implementation can be found in the `maven.model` module. However, in practice more complex actions are needed, which complicates the `ComponentUpdater` implementation.

The following aspects need to be taken into account when implementing the `ComponentUpdater` interface.

1. Usually, a `XAM`-based model corresponds to a certain XML file, which is in turn based on XML Schema. The Schema element `<sequence>` requires placing child items in a certain order. This rule is often neglected in practice. However, when following this rule, a position of a new element should be determined during editing.
2. Usually elements of the same type are located in a specific place of a document according to the XML Schema. But there are no rigid rules of what places cannot contain certain elements. Therefore, the elements of the same type can have parents of different types. This aspect might complicate `ComponentUpdater`, because it can require different processing depending on the parent type.
3. Generally, a model has event broadcasting facilities. But the level of event details can greatly vary, depending on the demand. The base `XAM` model does not impose any restrictions on the level of event details. In a simple case, a model can be silent while the components are being added or deleted. If a `XAM` model client needs information, event broadcasting is usually embedded into the `ComponentUpdater` implementation. The more information is needed, the implementation of the updater becomes more complex.

In a complex implementation of `ComponentUpdater`, the Visitor Pattern approach is usually used. In this case, the implementation class implements both the `ComponentUpdater` interface and a Visitor interface, which is specific to the `XAM` model. The visitor uses the added component as the visited object. You can see an example of a complex Updater implementation in the Schema model module:

```
org.netbeans.modules.xml.schema.model.impl.xdm.SyncUpdateVisitor.
```

It is worth mentioning that the `ComponentUpdater` implementation can be vulnerable and might lead to losing synchronization between the `XDM` and `XAM` models. It is very difficult to detect and fix such errors. A usual reason of the mistake is an evolutionary extension of the `XAM` model. New components are added while the model is being extended. It is easy to forget adding special processing for the new components to `ComponentUpdater`. As a result, the `XAM` model is not updated when such components are modified in correspondence with the `XDM` model. Implementations based on Visitor Pattern are especially vulnerable because they do not throw any compilation errors.

The following approach is suggested in order to prevent such mistakes. The result of processing of each request has to be tracked explicitly in `ComponentUpdater`. If a request was ignored or even caused an error, a `RuntimeException` is to be thrown. The exception will interrupt synchronization and will cause reconstruction of the entire `XAM` model as it was [described above](#).

WSDL Model: Extensibility and Support for Embedded Models

A WSDL document can contain an embedded XML Schema. Moreover, a WSDL model can contain extension components, for example, `PartnerLinkType` declarations for BPEL. Such extensions are not parts of the main WSDL model and are declared externally. Support of various WSDL specifications is a WSDL model feature.

The base `XAM` model does not provide these capabilities. The WSDL model is a more advanced implementation of the base `XAM` model. If extensibility and support for nested models is required for your new mode, you can take the WSDL model as an example.

The extensibility and embedding support are implemented using similar mechanisms. In fact, providing support for embedded models can be treated as a special case of extensibility. Therefore, they can be discussed at the same time.

The following are the main steps implemented in the WSDL model for extensibility support:

- The `xml.wsd1.model.ExtensibilityElement` interface. It is used to describe extensions. Note that even native WSDL components are treated as extensions. It allows one to unify the components structure.
- The `EmbeddedModel` interface is a root component of an embedded model. It links the host model and embedded models. It is a special case of an extension, because it also extends `ExtensibilityElement`.
- The `xml.wsd1.model.spi.ElementFactory` interface. It is used to decentralize components construction. So they can be constructed in different classes and not only inside of the host model module, but also inside external modules. Moreover, this interface provides a set of component names (`Qname`) for which it is responsible.
- The `xml.wsd1.model.impl.ElementFactoryRegistry` class. This class registers implementations of `ElementFactory` and knows which factory is responsible for constructing a specific component. It can also provide information about all supported components including extensions. In particular, it is used to determine [domain spaces](#).

These mechanisms implemented in the WSDL model can also be used to improve the base XAM model.

Alphabetical index

class.....	
AbstractModelFactory.....	4
Change.....	8
ChangeInfo.....	6
ComponentUpdater.....	5
Difference.....	8
DocumentModelAccess.....	5
FlashVisitor.....	15
MergeDiff.....	18
ModelAccess.....	4
NodeIdDiffFinder.....	18
NodeInfo.....	7
SyncPreparation.....	16
SyncUnit.....	6
Transaction.....	5
XDMAccess.....	5
XDMListener.....	19
XDMMModel.....	7
XDMMModelUndoableEdit.....	7