Customizing the Topologi Markup Editor

Confidential

draft 2002-08-28

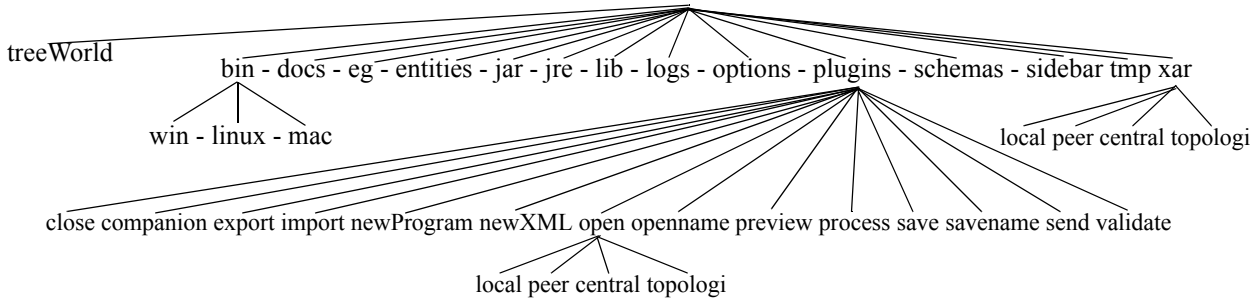
Customization

The *Topologi Markup Editor* provides several mechanism for customization:

- import, export, preview, validation, companion document and processing plug-ins;
- the services APIs: DB Input, Options, Messages.

The plug-in mechanism allows automatic updates over the WWW so that your editor has the most recent selection and version of plug-ins. It also allows you to customize plug-ins and conveniently deploy them to all the peers in a workgroup.

An important part of customizing the *Topologi Markup Editor* is to understand the file layout of the application. The application will typically be loaded on Windows under `c:\Program Files\Topologi\tme-nn-nn` where the `nn-nn` is a version number. This directory may contain various files relating to installating and running the editor.



Underneath are various directories:

- `bin` holds executable programs called by plugins;
- `docs` holds documentation;
- `eg` holds example files;
- `entities` holds standard entity files;
- `jar` holds the Java jar files that make up the application;
- `jre` holds the Java distribution and the main libraries for the application (JDK 1.3.1);

- `lib` holds external files that are needed by the application (e.g XSLT stylesheets);
- `logs` holds the log files that are generated by the application;
- `options` holds data used by the options system;
- `plugins` holds the plug-in BeanShell scripts;
- `schemas` holds various schema language files and processing stylesheets (probably unzipped from XAR files);
- `sidebar` holds the files that defines what should appear in the SideBar if this function is used in the editor (these files are created from XAR files);
- `tmp` holds temporary files used by the application. Most of these files will be removed on application shutdown;
- `xar` holds XAR files;
- `treeWorld` holds directories relating to TreeWorld peer services. This version of this manual does not give details of these directories.

PLUG-INS

A plug-in is a .bsh BeanShell script, perhaps with some accompanying .jar JAR files, which, when copied to the specific directory of the Topologi Markup Editor installation directory, becomes available to the users.

When you open a .bsh file, the editor will start up a Java Class browser. This provides some help in tracking down the names and signitures of available classes and methods. To help track down syntax errors, the editor will color-code the Java code. If you select *Check>Validate>Well-Formed* the editor will perform some other syntactical checking, in particular that parentheses and braces are balanced.

Names and Locations

A plug-in filename is made from four parts: (`number` “)”)? `name` “-” `version`“.” `extension`.

- The optional `number` is used for sorting names, to get a nice order in the menus
- The `name` is a string that is used for presentation or matching. It should not contain any whitespace; any “_” will be presented to the user as a space.
- The `version` is a string. It comes after a “-” character. If there are several plug-in scripts with the same name, then one with the largest version string (compared using string operations) will be used.
- The `extension` must be “`bsh`”.

These plug-ins can be created in the editor, and loaded merely by placing them in the appropriate directory. If your plug-in is not working correctly, it should not crash the editor and can be removed by simply deleting the file. However, the BeanShell has weak error-reporting. The various plug-ins provided can act as models so that you can create some new scripts.

The directories in which a plug-in .bsh script can be placed are under the following:

```
import/
export/
validate/
preview/
process/
send/
newXML/
newProgram/
open/
save/
openname/
savename/
companion/
```

Each of these directories has several subdirectories:

```
local/
peer/
central/
topologi/
```

The editor first uses the plug-ins with the most recent version in the `local/` directory, then the `peer/` directory, then the `central/` directory, then the `topologi/` directory. The `topologi/` directory will be updated with the latest versions of plug-ins from the Topologi webserver automatically; if for some reason a new plug-in is automatically downloaded that does not work, you can use the previous plug-in by copying the plug-in with the second largest version from the `topologi/` directory to the `central/` `peer/` or `local/` directory. The `central/` directory will be updated with the latest versions of plug-ins from your own webserver automatically; if for some reason a new plug-in is automatically downloaded that does not work, you can use the previous plug-in by copying the plug-in with the second largest version from the `central/` directory to the `peer/` or `local/` directory. The `peer/` directory will be updated with any plug-ins found on peer computers automatically; if you add a plug-in to the peer directory, it will automatically find its way to other peers in the same workgroup.[1]

Automated update and server locations can be set by the user using the Options system.

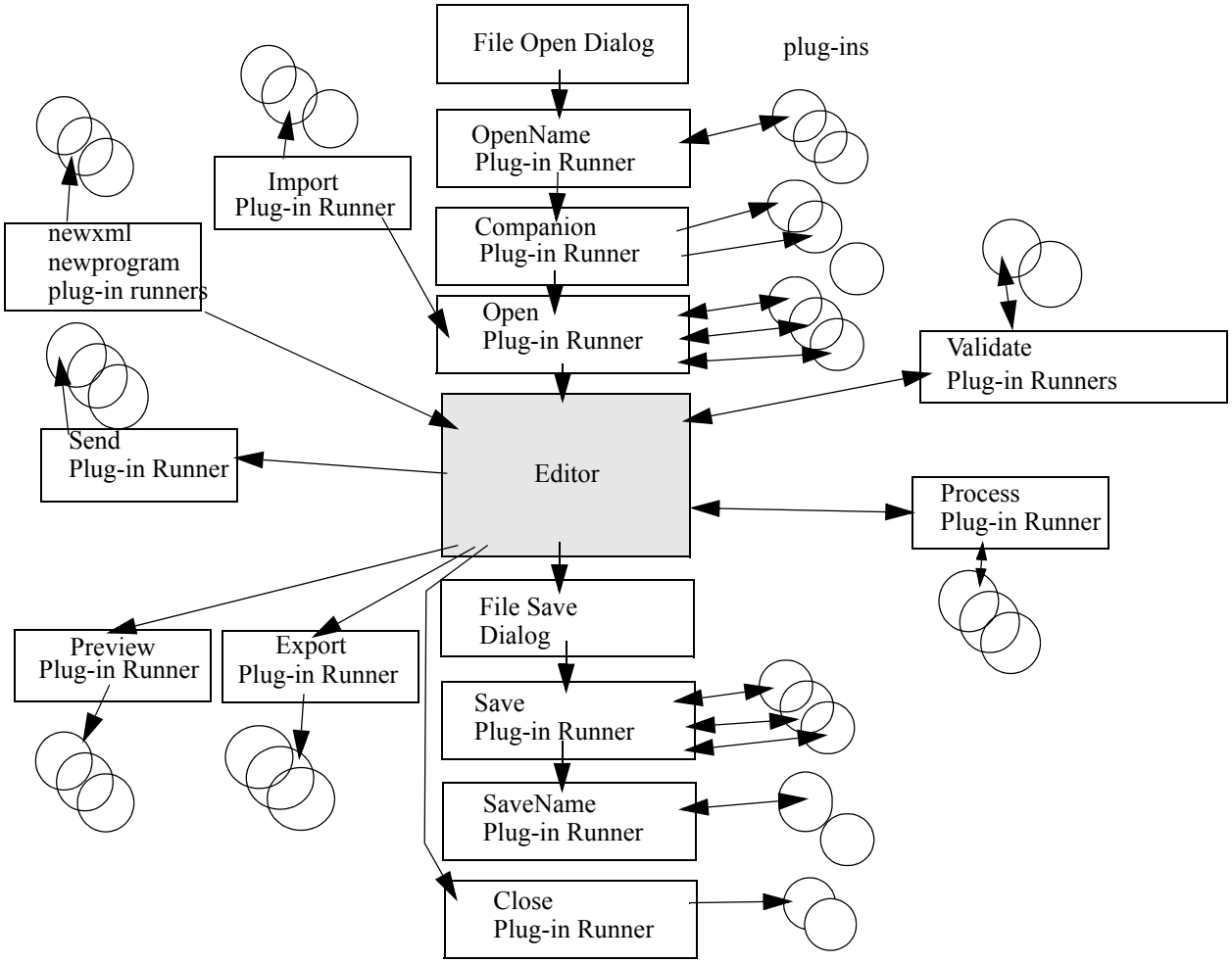
For `import/*`, `export/*`, `validate/*`, `preview/*`, and `process/*`, the *Topologi Markup Editor* looks at all the `.bsh` files in these directories, strips off the file extension, and uses the rest of the name in the appropriate menus for the user. For example, adding a new script "`Martian-001.bsh`" into the `import/local/` directory would mean than, the next time the *Topologi Markup Editor* is invoked, the Import menu will have a menu item *Martian* added.

1. To be clear, here is an example. Say we have the plug-ins for importing RTF in the files : `plugins/import/topologi/RTF-001.bsh`, and `plugins/import/peer/RTF-002.bsh`. The plug-in in the `peer/` directory will be used. If there is an automatic update, and the file `plugins/import/topologi/RTF-003.bsh` comes from the Topologi website, then the plug-in from the `peer/` directory will still be used, because it has priority despite the version number. If I put a file `plugins/import/local/RTF-002.bsh` then it has priority.

For `open/*`, `save/*`, `openname/*`, `savename/*`, the *Topologi Markup Editor* looks at the `.bsh` files in these directories, strips off the file extension, and uses the rest of the name to match the extensions of files being opened or closed, etc. The script is run against the file being opened or closed, etc. This allows you to customize how a file is pre-processed on open or close, etc., based on its file extension only.

For `companion/*`, the *Topologi Markup Editor* will check whenever a file is opened if the opened file has some companion file. A companion file is a file in the same directory with the same name as the file being opened and an extension that match the name of one of the companion plugin scripts. If this is the case then the script for that extension will be run on the companion file. Example, say that we open a file called `odrl.xml` and that there is a file called `odrl.ext` in the same directory. If there also exists a plug-in script in the companion directory with the name `ext.bsh` then this script will be run and the location of the companion file (`odrl.ext`) will be passed as a parameter to the script.

The plug-ins run the following order:



The *Topologi Markup Editor* uses file extensions to determine the type of the file, using the open plugin scripts. If the extension is unknown, the file will be opened as XML. In XML files, it will use any <?xar?> processing instrauction(see below) to look for the appropriate XML Application Archive (a kind of ZIP file which bundles together DTDs and stylesheets). If you have opened the file as an XML file, without any XAR, it will attempt to use whatever doctype, schemaLocation or namespace declarations are present when attempting to validate.

The editor will, at the last stage before opening or importing data (i.e., after all plug-ins have been run), allow character set conversion and newline handling: a dialog box will open to allow the user to provide details. When saving a file, all the save plug-ins are run first on the data. If the save plug-in script converts the data to a binary representation then this data will be saved. If the plug-in script doesn't create a binary representation of the data then a dialog box will open to allow the user to provide character set conversion and newline handling before the data is saved. After the data has been saved to a file the savename plug-ins are run if a script exists for the filename of the saved file.

See Encoding below

Framework

All plug-ins have the same Java interface:

```
public interface PlugInScript {

/**
 * This method should contain the code that performs the tasks of the plug in.
 * When the plug in has finished it should call the handleScriptResult(PlugInData pid)
 * method on the PlugInResultHandler passed as a parameter.
 * @param: pd The PlugInData passed to the script
 * @param: handler The PlugInResultHandler that should handle the script result
 */
public void executePlugIn(PlugInData pd, PlugInResultHandler handler);
/**
 * Method that determines when this plug in script should be enabled.
 * @param: td The TopologiDocument that is currently active in the editor
 * @return: true if the plug in script should be enabled based on the information
 * in the TopologiDocument. Just return true if the plug in script should always
 * be available.
 */
public boolean isEnabled(TopologiDocument td);
}
```

In a BeanShell script, the syntax for implementing this interface is a little different from standard Java; however, apart from this, the full syntax of Java can be used.

```
// Copyright 2001, Topologi Pty. Ltd.
// This file is part of the Topologi Markup Editor distribution

import java.io.*;
import com.topologi.tme1.io.plugins.*;
import com.topologi.tme1.io.worklistmodel.*;
import com.topologi.tme1.services.statusbar.*;

executePlugIn(PlugInData p, PlugInResultHandler handler) {

   if (p.getData() == null || p.getData().toString().equals("")) {
      p.getStatusLine().handleMessage(new StatusMessage("No data was sent to the script", StatusMessage.TIP, Sta-
     tusMessage.GENERAL_SERVICE, StatusMessage.GENERAL_PRIORITY, (short) 50));
      handler.handleScriptResult(null);
```

```
        return;
    }

    String outString = "<![CDATA[" + p.getData().toString() + "]]>";
    p.setData(new StringBuffer(outString));
    handler.handleScriptResult(p);
}

boolean isEnabled(TopologiDocument td) {
    return true;
}

return (PlugInScript) this;
```

So a plug-in is a script in a Java dialect which creates a runnable object. This object, when initialized with particular `PlugInData` and run, acts as a filter on the binary or text data, or on various kinds of metadata, coming in or out of the editor.

TreeWorld plug-ins, which implement services for external peers, use an extended version of the same interface:

```
public interface ServicePlugInScript {

    /**
     * Get the status of the this service action
     * For now the status is used as follow:
     *        first item in array is the description of the action
     *        (used in the popup menu in the client)
     *        second item is the action type, can be of two types: 'embed' or 'replace'
     * @return: String array of different status types
     */
    public String[] getStatus();
    /**
     * Execute the Plugin script
     * @param: pd The PlugInData
     * @param: handler The handler we report to when done
     */
    public void executePlugIn(PlugInData pd, PlugInResultHandler handler);
    /**
     * Return true if the script is enabled
     * @return: true if enabled
     */
    public boolean isEnabled();
}
```

In TreeWorld plugIns, if `isEnabled()` returns false, that service will not appear on the menus of TreeWorld browsers, however the services is still executable using the URL. This version of this manual does not give details on TreeWorld plugins.

PlugInData

The filter is provided, by the plug-in framework, a `PlugInData` object containing the filenames or data for use as input, and a callback by which the data, when ready, is returned. The `PlugInData` object has the following declaration:

```
/**
 * Generic data object used for input and output of plug-ins.
 * The DocumentData object that is a property of each document in the editor is always passed through the
 * PlugInData object to the plug in scripts. When the result comes back from the script the data in
 * DocumentData together with the other properties in the PlugInData object are processed and the
 * appropriate action is taken depending on the PlugInResultHandler that handle the result.
 */
public final class PlugInData {
    /**
     * Input and output of text.
```

```
 */
private StringBuffer data;
/**
 * If something goes wrong this property can be used to pass errors from
 * plug in scripts.
 */
private StringBuffer error;
/**
 * Pass filename to/from a plug in script
 */
private String filename;
/**
 * The Services object used to access all the services provided
 */
private Services services;
/**
 * Various meta data about a document. Each document open in the editor will
 * have a DocumentData object.
 */
private DocumentData documentData;
 /**
 * Exchange binary data between plug in scripts and the PlugInHandler
 */
private byte[] binary;
/**
 * The origin of the PlugInData (Open, Process, Import etc)
 */
private String origin;
/**
* The status of this PlugInData. By default this is always OK.
*/
private short status = OK;

/**
* Status OK
*/
public static final short OK = 0;
/**
* Status ERROR
*/
public static final short ERROR = 1;
/**
* Status CANCELLED
*/
public static final short CANCELLED = 2;
/**
 * Open origin
 */
public static final String OPEN = "Open";
 /**
 * Process origin
 */
public static final String PROCESS = "Process";
/**
 * Import origin
 */
public static final String IMPORT = "Import";
/**
 * Close and Save origin
 */
public static final String CLOSESAVE = "CloseAndSave";
/**
 * Close
 */
public static final String CLOSE = "Close";
/**
```

```
 * Save origin
 */
public static final String SAVE = "Save";
/**
 * Validation origin
 */
public static final String VALIDATION = "Validation";
/**
 * New origin
 */
public static final String NEW = "New";

/
  ***************************************************************************************
  *****/
/* ---------------------------------------- Public methods ----------------------------------------*/
/
  ***************************************************************************************
  *****/
public synchronized byte[] getBinary()
public synchronized StringBuffer getData()
/**
 * This method will return the DocumentData object. It's worth noting that this
 * will always return an object and never return null which means it's always
 * safe to do: pid.getDocumentData().put("companion.filename", filename);
 * @return: The DocumentData object
 */
public synchronized DocumentData getDocumentData()
public synchronized StringBuffer getError()
public JFileChooser getFileChooser()
public synchronized String getFilename()
public OptionsFrameWork getOptions()
public synchronized String getOrigin()
public synchronized Services getServices()
public short getStatus()
public StatusBarInterface getStatusLine()
/**
 * Method that transcode text data into binary data. It checks if the PlugInData has text data and converts
 * it into binary data in the binary field. It also checks if the documentData has the property
 * DOCUMENT_EXPORT_TRANSCODING_SETTINGS which is present if the data has already been
 * exported and converted to binary. If the property exists, it retrieves the settings as a
 * TranscodingResultat. It uses the TextConverter to transcode the data.
 * This method does not provide the owner of the window. The window will then be a standard Window.
 * It is used when the encoding is needed from a beanshell script.
 * @param: encoding the default encoding used, the final one is chosen using the window
 * @param: transcodeXML if true, XML delimiters are transcoded (& and <)
 * @param: aboveChar int that represents the tide number for which characters to convert to XML
 * character references
 * @param: lineLength the maximum line-length before wrapping
 * @return: false if an error ocurred, true otherwise
 */
public boolean guaranteeBinary(String encoding, boolean transcodeXML, int aboveChar,
   int lineLength)
/**
 * A call to this method will guarantee that the correct DOCTYPE declaration is
 * propagated or inserted in the PlugInData.data property. The general algorithm
 * is as follows:
 * If DocumentData.DOCUMENT_DTD_LOCATION is set then
 * Create a new DOCTYPE declaration based on the first XML Element in the data
 * and the system identifier stored in DocumentData.DOCUMENT_DTD_LOCATION. If a
 * DOCTYPE declaration already exist this is replaced by this new one otherwise
 * the DOCTYPE will be placed before the first XML Element.
 * Else
 * If a DOCTYPE is present in the data then use that
 * otherwise check the DocumentData.DOCUMENT_DOCTYPE property and see if a
 * DOCTYPE has been saved.
 * If this is the case then replace the document element in this declaration with
```

```
     * the name of the first XML element in the data and insert the DOCTYPE
     * declaration before this first element.
     */
    public void guaranteeDOCTYPE()
    /**
     * Guarantee that namespace declarations are carried forward if a selection
     * is made in the document. If no selection is made then nothing is done since
     * nothing can be carried forward. If a selection is made the namespace
     * declarations in DocumentData.DOCUMENT_NAMESPACE_DECLARATIONS will be inserted
     * in the start tag of the first XML Element in the selection.
     */
    public void guaranteeNamespaceDecl()
    /**
     * Method that transcode binary data into text data. It checks if the PlugInData
     * has binary data and converts it into text in the data field. If no binary data
     * is present, it checks for the filename and converts the file if there is one.
     * It uses the TextConverter to transcode the data.
     * This method provides the owner of the window as a parameter.
     * The window will then be a Dialog and this will lock the owner window. It is used
     * when the encoding is needed within the code, not from a beanshell script.
     * Creation date: (11/19/2001 5:30:26 PM)
     * @param: encoding the default encoding used, the final one is chosen using the window
     * @param: transcodeXML if true, XML delimiters are transcoded (& and <)
     * @param: aboveChar int that represents the tide number for which characters to convert to XML
     * character references
     * @param: lineLength the maximum line-length before wrapping
     * @return: false if an error ocurred, true otherwise
     */
    public boolean guaranteeUnicode(String encoding, boolean transcodeXML, int aboveChar,
        int lineLength)
    /**
     * Guarantee that the DocumentData.VALIDATION_SCHEMALOCATION and
     * DocumentData.VALIDATION_NONAMESPACESCHEMALOCATION properties gets set to
     * the correct values if a schema is available from somewhere. If the
     * DocumentData.DOCUMENT_SCHEMA_LOCATION property is set then this schema will be
     * used and parsed to find the correct targetNamespace if one exists.
     */
    public void guaranteeXMLSchema()
    /**
     * Check if the value of the data property is well-formed XML.
     * @return: true if the data is well-formed, otherwise false
     */
    public boolean isWellFormed()
    public synchronized void setBinary(byte[] binary)
    public synchronized void setData(String data)
    public synchronized void setData(StringBuffer data)
    public synchronized void setDocumentData(DocumentData docData)
    public synchronized void setError(String error)
    public synchronized void setError(StringBuffer error)
    public synchronized void setFilename(String filename)
    public synchronized void setOrigin(String o)
    public synchronized void setServices(Services services)
    /**
     * Set the status of this <code>PlugInData</code> object.
     * Can be one of the following values: <code>PlugInData.OK</code>,
     * <code>PlugInData.ERROR</code> or <code>PlugInData.CANCELLED</code>
     * @param: s The new status
     */
    public void setStatus(short s)
}
```

A plug-in should never create its own PluginData object. It should use the object provided and should ALWAYS return a valid PlugInData object when it makes a call to the handler.

Each plug-in may load a jar file, which should have the same name and location. The relationship between versions of the jars and versions of the scripts is up to developers to maintain: it is not managed. If you call your jar file by a different name to a plug-in, it will not be propagated between systems. If you use the same name as another plug-in, then that plug-in's jar file will be overwritten. So it is best to use the same name. Use the `addClasspath("xxx.jar")` command in the script. Here is an example where an external jar file is used to validate RELAX-NG schemas:

```
// Copyright 2001, Topologi Pty. Ltd.
// This file is part of the Topologi Markup Editor distribution
import validation.relaxng.*; // This package is contained in the external jar file (RelaxValidator.jar)
import com.topologi.tme1.io.plugins.*;
import com.topologi.tme1.io.worklistmodel.*;
import com.topologi.tme1.services.statusbar.*;

executePlugIn(PlugInData  p, PlugInResultHandler handler)  {

    // handle bad data
    if (p.getData() == null || p.getData().toString().length() == 0)  {
        p.setError("No data received for validation");
        handler.handleScriptResult(p);
        return;
    }

    String sep = System.getProperty("file.separator");
    // Add the RelaxValidator.jar file to the class path
    addClassPath(p.getOptions().getPluginDir() + sep + "validate" + sep + "topologi" + sep +
    "RelaxValidator.jar");
    // Create the validator that validates the file
    RelaxValidator validator = new RelaxValidator(handler, p);
    // Start the validator which in this case is a separate thread. When it finishes it will
    // make a call to handler.handlePlugInResult(p);
    validator.validate();
}

boolean isEnabled(TopologiDocument td) {
    // Notation must be XML
    if (!td.getNotation().equals("xml")) {
        return false;
    }
   // Make sure we have a RELAX-NG schema specified for the current document
   if (!td.getDocumentData().hasDataValue(DocumentData.DOCUMENT_RELAX_LOCATION)) {
        return false;
    }
    return true;
}

return (PlugInScript)this;
```

DocumentData

The `documentData` field is a hash table. It is used to store arbitrary data which one plug-in may send to another and to communicate different properties to and from the document that is open in the editor. (To store or access data between invocations of the same plug-in, or between different plug-ins, use the `sessionData` hash table of the options framework.)

To prevent nameclashes, please use the following conventions when forming your own key: *plugindir.extension.dataname*, such as `import.svg.screenratio`.

Dublin Core Keys. The Dublin Core (http://dublincore.org/documents/dces/) defines a set of useful names for metadata, which plugins should use where possible, for maximum interoperability. Here are key strings for using the Dublin Core metadata and the HTTP MIME headers (note, case-sensitive keys; lists are allowed for round-tripping purposes, however the first string in the list will be the one selected when only one metadatum is required.):

- "dc:identifier" Vector of String, Dublin Core metadata for URI for the file;
- "dc:date" Vector of String, Dublin Core metadata name for (creation or publishing) date YYYY-MM-DD;
- "dc:source" Vector of String, Dublin Core metadata name. Plugins can use it for the stacking the various intermediate filenames obtained by open scripts, for use by the corresponding save script, e.g. when x.xml is derived from x.zip; Used to store different sources.
- "dc:title" Vector of String, Dublin Core metadata name for formal name (e.g. SGML FPI);
- "dc:creator" Vector of String, Dublin Core metadata name for creator;
- "dc:subject" Vector of String, Dublin Core metadata name for subject, including keywords;
- "dc:description" Vector of String, Dublin Core metadata name for human readable description;
- "dc:publisher" Vector of String, Dublin Core metadata name for the publisher of the data;
- "dc:contributor" Vector of String, Dublin Core metadata name for contributor;
- "dc:type" Vector of String, Dublin Core metadata name for general document type;
- "dc:format" Vector of String, Dublin Core metadata for MIME type;
- "dc:language" Vector of String, Dublin Core metadata for the natural language of a text, such as "en-AU";
- "dc:relation" Vector of String, Dublin Core metadata for some related resource
- "dc:coverage" Vector of String, Dublin Core metadata for geographical or time coverage information
- "dc:rights" Vector of String, Dublin Core metadata for rights information. This may be an ODRL document in XML, a URL (e.g. to an ODRL document) or other text.

In order to simplifiy the entry of Dublin Core metadata (which is a vector of strings, since one kind of metadata may have multiple instances) there is a simple API:

```
void setDCProperty(String key, String value)
String getDCProperty(String key)
String deleteDCProperty(String key)
void replaceDCProperty(String key, String value)
void setDCStack(String name, Stack dcStack)
Stack getDCStack()

PlugInData_object.getDocumentData().setDCProperty("dc:source", source_id)
```

Although the above will work it's safer to use the public constants to access the predefined properties of the `DocumentData` object. In the above case this would be:

```
PlugInData_object.getDocumentData().setDCProperty(DocumentData.DC_SOURCE, source_id)
```

If you do it this way then you will always get the correct key value and you don't have to worry about spelling errors that will cause unexpected results. All the predefined key values can be accessed in a similar way from the `DocumentData` class and for a full listing of these keys and more documentation on the above methods see the `DocumentData` listing at the end of the section.

File Information Keys. The following keys are available to set up the correct editing mode or facilities for an extension or file (most are dealt with in the section on `open/` plug-ins below):

- "file.readonly" - If a file being opened is readonly or not. The type is a `String` with the valid values: `true` or `false`.

  String readOnly = (String)docData[2].getProperty(DocumentData.FILE_READONLY);

- "file.date" - The last modification date for the file being opened. The type is a `String` in the format "dd/MM/yyyy hh:mm [AM|PM]"

  String date = (String)docData.getProperty(DocumentData.FILE_DATE);

Document Information Keys. These fields can contain information used to set up a document and process it.

- "document.notation" - The notation the document has (should have). The type is a `String` and can have one of the following values: `text`, `xml`, `dtd`, `sgml`, `bat`, `c++`, `eiffel`, `idl`, `java`, `javascript`, `php`, `tex`, `sql`, `c`

  String notation = (String)docData.getProperty(DocumentData.DOCUMENT_NOTATION);

- "document.hasfile" - True if the document has a corresponding file on the filesystem which means that the document has either been saved at least once or the document was opened from a file. The type is a `String` with the valid values: `true` or `false`.

  String hasFile = (String)docData.getProperty(DocumentData.DOCUMENT_HASFILE);

- "document.isdirty" - True if the document has been modified since it was last saved. The type is a `String` with the valid values: `true` or `false`.

  String isModified = (String)docData.getProperty(DocumentData.DOCUMENT_ISDIRTY);

- "document.xar.filename" - If the document has a XAR attached to it the name of the XAR will be stored in this property. The type is a `String`

  String xarName = (String)docData.getProperty(DocumentData.DOCUMENT_XAR_FILENAME);

- "document_selection" - True if a selection was made in the document when a script was called. The type is a `String` with the valid values: `true` or `false`.

  String haveSelection = (String)docData.getProperty(DocumentData.DOCUMENT_SELECTION);

- "document_selectionData" - The SelectionData object with information about the selection. The type is `SelectionData` (See description in the SelectionData section)

  ```
  SelectionData sel = (SelectionData)docData.getProperty(
     DocumentData.DOCUMENT_SELECTIONDATA);
  ```

2. All these example requires that you have created `DocumentData` object called docData. E.g.
   DocumentData docData = PlugInData_object.getDocumentData();

- "document.oldname" - The absolute name of the old filename that was used before a new name was selected in for example the SAVEAS dialog. The type is a `String`

  ```
  String oldName = (String)docData.getProperty(DocumentData.DOCUMENT_OLDNAME);
  ```

- "document_doctype" - The DocType object describing the DOCTYPE declaration if one exists in the document. The type is `DocType` (See description in the DocType section)

  ```
  DocType docType= (DocType)docData.getProperty(DocumentData.DOCUMENT_DOCTYPE);
  ```

- "document_namespace_declarations" - The namespace declarations that exists in the document up until the selection starts if a selection is made in the document. If no selection is made this property won't exist. The type is a Java `Map` interface

  ```
  Map ns = (Map)docData.getProperty(DocumentData.DOCUMENT_NAMESPACE_DECLARATIONS);
  ```

- "encoding" - The encoding the document was saved with. The type is a `String`

  ```
  String encoding = (String)docData.getProperty(DocumentData.DOCUMENT_ENCODING);
  ```

The following properties all define an external file that is associated with a document. Typically these are all set *en masse* by reading an XAR file.

- "dtd-location" - The location of a DTD
- "sgml-dtd-location" - The location of a SGML DTD
- "css-location" - The location of a CSS stylesheet
- "index-location" - The location of an Index file
- "relax-location" - The location of a RELAX-NG schema
- "schematron-location" - The location of a Schematron schema
- "schema-location" - The location of a W3C XML Schema
- "xsl-location" - The location of an XSL stylesheet
- "catalog-location" - The location of a Catalog file

E.g.

```
String xsl = (String)docData.getProperty(DocumentData.DOCUMENT_XSL_LOCATION);
```

Plugin-specific Keys. The following properties contain information that are used in the context of certain types of plugins.

- "companion.filename" - Used to store the absolute filename of the companion file. The type is a `String`

  ```
  String companionName = (String)docData.getProperty(DocumentData.COMPANION_FILENAME);
  ```

- "open.zip.tempfile" - Used to store the absolute filename of the temporary zip file being created if we have zips within zips. The type is a `String`

  ```
  String zipFile = (String)docData.getProperty(DocumentData.OPEN_ZIP_TEMPFILE);
  ```

- "save.previous.filename" - Used to store the name of the file that was last checked for a plugin script if the DC_SOURCE property has multiple values when a document is being saved. The type is a `String`

  ```
  String prevName = (String)docData.getProperty(DocumentData.SAVE_PREVIOUS_FILENAME);
  ```

- "save.original.filename" - Used to store the original name of the document as it appears in the editor when a document is saved. The type is a `String`

  ```
  String origName = (String)docData.getProperty(DocumentData.SAVE_ORIGINAL_FILENAME);
  ```

- "preview.original.filename" - Used to store the original name of the document as it appears in the editor when executing the process script. The type is a `String`

  ```
  String origName = (String)docData.getProperty(DocumentData.PREVIEW_ORIGINAL_FILENAME);
  ```

- "import.url.headers" - Used to store all the headers received from the Webserver when importing data from a URL. This object is another Java `java.util.Hashtable` with keys and values received from the `java.net.URLConnection` object.

  ```
  keys = URLConnection.getHeaderFieldKey(i)
  values = URLConnection.getHeaderField(i)
  ```

  The type is `Hashtable`.

  ```
  Hashtable mime = (Hashtable)docData.getProperty(DocumentData.IMPORT_URL_HEADERS);
  String date = (String) mime.get("Date");
  String contentLength = (String) mime.get("Content-length");
  String server = (String) mime.get("Server");
  String filterRevision = (String) mime.get("Filter-Revision");
  String contentType = (String) mime.get("Content-Type");
  String lastModified = (String) mime.get("Last-Modified");
  String connection = (String) mime.get("Connection");
  ```

- "validation.schemalocation" - The schemaLocation values if a W3C XML Schema with a namespace is used for validation. The format is the same as for the `schemaLocation` attribute in the W3C XML Schema document, i.e a space separated list of namespace and schema location URI pairs. For example:

  ```
  "www.test.com/schema c:\test.xsd".
  ```

If this property is modified it's important to check if it already has a value or not. If it does then the new value should be appended to the end of the old value with a space as a separator. The type is `String`

```
String schemaLoc = (String)docData.getProperty(
   DocumentData.VALIDATION_SCHEMALOCATION);
```

- "validation.nonamespaceschemalocation" - The noNamespaceschemaLocation value if a W3C XML Schema with no targetNamespace is used. The type is `String`

  ```
  String noNamespaceSchemaLoc = (String)docData.getProperty(
     DocumentData.VALIDATION_NONAMESPACESCHEMALOCATION);
  ```

- "validation.errors" - Used to store a list of all the errors. Each item in the list should have the following `String` value (each property is separated by the character sequence ‘{||}’):

  ```
  “errortype{||}filename{||}linenumber{||}columnnumber{||}message”
  ```

Where errortype currently can have the following values:

```
(ValidationMessage.ERROR | ValidationMessage.FATALERROR | ValidationMessage.WARNING |
ValidationMessage.MESSAGE | ValidationMessage.OK)
```

This property is used to report validation messages back from the Validation plugins and will be shown in the statusbar of the editor. The type is `List`

```
List errors = new LinkedList();
```

```
errors.add(new String("ValidationMessage.ERROR{||}c:\test\test.xml{||}2{||}10{||}Invalid attribute value"));
errors.add(new String("ValidationMessage.ERROR{||}c:\test\test.xml{||}4{||}1{||}Invalid element"));
docData.setProperty(DocumentData.VALIDATION_ERRORS, errors);
```

- "validation.no-error" - A message that should be shown in the editor if the file didn't contain any validation errors[3]. The type is `String`

  ```
  docData.setProperty(DocumentData.VALIDATION_NOERROR, "XML data is valid");
  ```

- "validation.option" - The option for the validation. In the editor there are two menu-items for validation, "Validate" and "Validate (Full Checks)". In the validation scripts this option can be used to check from which menuitem the script was called. The option can have two values:

  ```
  (ValidationPlugInHandler.FULL | ValidationPlugInHandler.NORMAL)
  ```

  The type is `String`

  ```
  if (((String)docData.getProperty(DocumentData.VALIDATION_OPTION)).equals(
  ValidationPlugInHandler.NORMAL)) {
     // Do normal validation
  } else {
     // Do full validation
  }
  ```

- "validation_error_offset" - The offset of the error messages if a selection has been made in the document or the data has been otherwise modified before validation (for example if a DOCTYPE declaration has been added at the top if the selected text). The type is `OffSet` (See the OffSet section for more information).

For example if a line has been added at the start of the selection the following code would modify the offset to accomodate this new line:

```
// Update the Error offset to accomodate the new line
OffSet offset;
if (docData.hasDataValue(DocumentData.VALIDATION_ERROR_OFFSET)) {
   offset = (OffSet) docData.getProperty(DocumentData.VALIDATION_ERROR_OFFSET);
   offset.setLine(offset.getLine() - 1);
} else {
   offset = new OffSet(-1, 0);
}
docData.setProperty(DocumentData.VALIDATION_ERROR_OFFSET, offset);
```

Boolean Keys. If you want to modify or compare a property which has a boolean value you can use the predefined boolean properties on the `DocumentData` object.

- "true" - Constant used for the boolean value `true`. The type is `String`

  ```
  docData.setProperty(DocumentData.DOCUMENT_SELECTION, DocumentData.TRUE);
  ```

- "false" - Constant used for the boolean value `false`. The type is `String`

  ```
  docData.setProperty(DocumentData.DOCUMENT_SELECTION, DocumentData.FALSE);
  ```

The methods accessible on the `DocumentData` object are as follows:

```
package com.topologi.tme1.io.worklistmodel;
```

3. NOTE: "**validation.errors**" and "**validation.no-error**" are mutually exclusive so if one exists the other cannot exist. In the code "**validation.no-error**" will have presedence so if this key exist the message will be shown in the Editor and "**validation.errors**" will not be checked at all.

```
/**
 * The DocumentData class contain all sorts of information about a
 * document in the form of properties. Each property is identified by a name,
 * value pair where the name must be unique and is used as the key in
 * the Hashtable where they are stored.
 * Properties are set and retrieved using the setProperty and getProperty methods.
 * Dublin Core metadata is also supported in the document and each Dublin Core (DC)
 * property has a unique name with the prefix dc: and the value is stored as a
 * Stack of Strings. This means that each DC property can have more than one value.
 * The DC properties are accessed using the xxxDCxxx methods.
 * The method hasDataValue can be used to query the DocumentData
 * object if a specific property exists and is non null. If the property is a
 * String or StringBuffer hasDataValue will also return false if the length is 0.
 */
public final class DocumentData {
    /**
     * Query and delete a Dublin core property. If the property exist the first item
     * in the Stack is removed and the value will be returned. Otherwise
     * it will return null.
      * @param: name The name of the DC property
     * @return: The String value of the property if it exists otherwise null
     */
    public synchronized String deleteDCProperty(String name)
    /**
     * Query a Dublin core property. If the property exist the top item on the
     * Stack will be returned otherwise it will return null.
      * @param: name The name of the DC property
     * @return: The String value of the property if it exists otherwise null
     */
    public synchronized String getDCProperty(String name)
    /**
     * Get the whole Stack for a DC property.
      * @param: name The property name
     * @return: The Stack if the property name has a Stack object
     * otherwise null
     */
    public Stack getDCStack(String name)
    /**
     * Query a property. If the property exist the value will be returned
     * otherwise it will return null.
      * @param: name The name of the property
     * @return: The Object for the property if it exists otherwise null
     */
    public synchronized Object getProperty(String name)
    /**
     * Helper method that that returns true if a property with a
     * specified key exists. If the Object stored in the
     * Hashtable is a String or StringBuffer,
     * true is only returned if the value is a non empty-string.
     * If the Object is a Stack then true is
     * only returned if the Stack isn't empty.
      * @param: name The name of the property to check
     * @return: see above
     */
    public final boolean hasDataValue(String name)
    /**
     * Remove a property value. This method can be used for both normal properties
     * and DC properties.
      * @param: name The name of the property to remove
     */
    public synchronized void removeProperty(String name)
    /**
     * Replace a value for a Dublin core property.
     * If the property already exists the old Stack will be removed
     * and a new Stack created with only one item which is the new
```

```
     * value.
     * @param: name The property name
     * @param: value The property value
     */
    public synchronized void replaceDCProperty(String name, String value)
    /**
     * Set a value for a Dublin core property.
     * If the property already exists the new value will be pushed ontop of the
     * Stack. If the property doesn't exist a new Stack
     * will be created and the value will be pushed on the new Stack.
     * @param: name The property name
     * @param: value The property value
     */
    public synchronized void setDCProperty(String name, String value)
    /**
     * Set a Dublin core property by providing a precreated Stack.
     * If the DC property already exists the old Stack will be replaced with the
     * new Stack.
     * @param: name The property name
     * @param: dcStack The property stack
     */
    public synchronized void setDCStack(String name, Stack dcStack)
    /**
     * Set a value for a property.
     * If the property already exists the new value will replace the old value
     * @param: name The property name
     * @param: value The property value
     */
    public synchronized void setProperty(String name, Object value)
}
```

SelectionData

The `SelectionData` object can be used in the PlugIn scripts to access the information about a selection that is made in the document. The `SelectionData` has the following declaration:

```
package com.topologi.tme1.io.plugins;

/**
 * Contains the line number and column position of the selection, if a selection
 * has been made in the document.
 */
public final class SelectionData {
    /**
     * The line number where the selection starts (the first line in the document is 0)
     */
    private int startLine;
    /**
     * The column position on the line where the selection starts (the start of the line is 0)
     */
    private int colPos;
    public int getColPos()
    public int getStartLine()
}
```

DocType

The `DocType` object can be used by the PlugIns to access the DOCTYPE declaration in the document if one exists. It's also possible to change the DOCTYPE in the document if this is desirable to do before for example validation. The `DocType` object has the following declaration:

```
package com.topologi.tme1.io.worklistmodel;
```

```
/**
 * Contains all the information about a DOCTYPE declaration. This object also lets
 * you modify the DOCTYPE declaration so that you can change the document element,
 * system identifier or public identifier.
 */
public final class DocType {
    /**
     * The text value of the complete DOCTYPE declaration
     */
    private String docType;
    /**
     * The name of the document element
     */
    private String docElement;
    /**
     * The value of the public identifier if it exists. Will be null
     * if none exist
     */
    private String publicId = null;
    /**
     * The value of the system identifier if it exists. Will be null
     * if none exist
     */
    private String systemId = null;
    /**
     * True if the DOCTYPE has an internal subset otherwise false
     */
    private boolean internal;
    /**
     * The start position of the DOCTYPE as it appears in the data when it's
     * extracted.
     */
    private int docTypeStart;
    /**
     * The end position of the DOCTYPE as it appears in the data when it's
     * extracted.
     */
    private int docTypeEnd;

    /**
     * Create a DOCTYPE declaration object with the DOCTYPE text.
      * @param: doc The DOCTYPE declaration string
     * @param: start Start location of the DOCTYPE in the data from which it is
     * extracted
     * @param: end End location of the DOCTYPE in the data from which it is
     * extracted
     */
    public DocType(String doc, int start, int end)
     public String getDocElement()
     public String getDocType()
     public boolean getInternal()
     /**
     * Return a modified DOCTYPE declaration which is modified with the new
     * values supplied to the function.
     * At this stage an existing DOCTYPE can only be modified with values that
     * already exist in the DOCTYPE. I.e, you can't add a systemId if a systemId
     * doesn't already exist.
     * If a null argument is passed as one of the parameters that property of the
     * DOCTYPE declaration isn't changed.
      * @param: newDoc The new Document element
     * @param: newSys The new System identifier
     * @param: newPub The new Public identifier
     * @return: The modified DOCTYPE declaration as a String
     */
    public String getModifiedDocType(String newDoc, String newPub, String newSys)
```

```
    public String getPublicId()
    public String getSystemId()
    /**
     * Set the name of the document element and the location where it appears
     * in the DOCTYPE declaration.
      * @param: el The name of the document element
     * @param: start Start location
     * @param: end End location
     */
    public void setDocElement(String el, int start, int end)
    /**
     * Set the whole DOCTYPE declaration.
      * @param: doc The DOCTYPE declaration string
     * @param: start Start location of the DOCTYPE in the data from which it is
     * extracted
     * @param: end End location of the DOCTYPE in the data from which it is
     * extracted
     */
    public void setDocType(String doc, int start, int end)
    public void setInternal(boolean i)
    /**
     * Set the Public identifier and the location where it appears
     * in the DOCTYPE declaration.
      * @param: pub The Public identifier
     * @param: start Start location
     * @param: end End location
     */
    public void setPublicId(String pub, int start, int end)
    /**
     * Set the System identifier and the location where it appears
     * in the DOCTYPE declaration.
      * @param: sys The System identifier
     * @param: start Start location
     * @param: end End location
     */
    public void setSystemId(String sys, int start, int end)
    public int getDocTypeStart()
    public int getDocTypeEnd()
}
```

OffSet

The `OffSet` object can be used by the PlugIn to modify the offset location used by the validation handler when it reports a validation problem with a line number. If, for example, a selection is made in the document and the PlugIn script adds a DOCTYPE declaration to the top of the selection then the `OffSet` object passed in the `DocumentData` object needs to be updated. The `OffSet` object has the following declaration:

```
package com.topologi.tme1.io.plugins;

/**
 * Contains a line offset and a column offset. This is mainly used for validation
 * when only a selection has been made in the document. In this case there is an
 * offset from where the selection starts in the document.
 * The column offset is currently unused but it's included for future use.
 */
public final class OffSet {
    /**
     * A line offset
     */
    private int line;
    /**
     * A column offset (currently unused)
     */
    private int col;
```

```
    /**
     * Create a new OffSet object.
     * @param: l The line offset
     * @param: c The column offset
     */
    public OffSet(int l, int c)
     public int getCol()
     public int getLine()
     public void setCol(int c)
     public void setLine(int l)
}
```

Here follows a brief description of what the PlugIn scripts in the different categories can be used for:

import/

An import plug-in provides some mechanism for fetching data from some other source. The script must provide all information, including selecting any files involved.

The data is written into the `data` field (a `StringBuffer`) of the `PlugInData` object if it is text.

```
pid.setData(new StringBuffer("Some text"));
```

If it is binary it should be stored as a `byte[]` array in the `binary` field.

```
pid.setBinary(new byte[] {...});
```

The plug-ins are available under the File>Import menu.

export/

The script takes text in the `data` field (a `StringBuffer`) of the `PlugInData` object and packages it in some way. The text is either was the currently selected text, if any is selected, or the whole file of the current file being edited. The script must provide all user interaction, including selecting any files involved.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PlugInData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

If you are packaging a group of files, each file must be exported separately.

The plug-ins are available under the File>Export menu.

validate/

You can add a document validation service to your document. The script takes text in the `data` field (a `StringBuffer`) of the `PlugInData` object. The text is either the currently selected text, if any is selected, or the whole of the current file being edited.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PlugInData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

The script can use the `DocumentData.VALIDATION_ERRORS` or the `DocumentData.VALIDATION_NOERROR` property of the DocumentData object to communicate errors to the *Topologi Markup Editor*. Otherwise, it must report to the user using pop-ups itself.

Make sure to remind the user to save to the file system the most recent versions of the subordinate entities of the document; otherwise the user may have one version of an entity in their editor while the validation results will be for the version in the file system. This could be confusing.

The plug-ins are available under the Edit>Validate menu when an XML document is being edited.

preview/

This lets you preview using some external typesetting system and browser. The script takes text in the `data` field (a `StringBuffer`) of the `PlugInData` object. The text is either the currently selected text, if any is selected, or the whole of the current file being edited.

If the preview plugin modifies the filename field of the `PlugInData` object so that it differs from the original filename (stored in `DocumentData.PREVIEW_ORIGINAL_FILENAME`) a new document will be created in the editor with this name. The data that is returned by the script in the `data` field of the `PlugInData` object will be inserted in this new document. Note: It is important that the new name is not a VALID filename because then conflicts may arise with the actual file on the filesystem. It should instead be something like "***Results from Preview script***".

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PluginData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

The plug-ins are available under the File>Preview menu.

send/

(This is reserved by not currently used.)

process/

This lets you process some text, for example as part of cut-and-paste. The script takes text in the `data` field (a `StringBuffer`) of the `PlugInData` object. The text is either the currently selected text, if any is selected, or the whole of the current file being edited.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PlugInData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

If the processing service reports some information, it is the plug-in's responsibility to report the information to the user.

The plug-ins are available under the Edit>Process menu.

newXML/

The `newXML` plug-ins provide the list on the main menu under New>XML>.

These plug-ins are basically very simple. The script must create an entry in the hash table `documentData` with the key "document.notation" and one of the following String values (if no notation is set explicitly, the editor defaults to XML):

- xml
- dtd
- sgml
- text

A custom newXML plug-in could, for example, put up a data form to read in meta data about the document, then generate an XML document with the metadata filled in. That way the user has convenient forms-based entry, if that was productive for the workers in question.

newProgram/

The `newProgram` plug-ins provide the list on the main menu under New>Programs>.

These plug-ins are basically very simple. The script must create an entry in the hash table `documentData` with the key "document.notation" and one of the following String values (if no notation is set explicitly, the editor defaults to XML):

- text
- bat
- c++
- c
- eiffel
- idl

- java
- javascript
- php
- tex
- sql

open/

The open plug-ins are checked whenever a file is to be opened. The script takes text in the `data` field (a `StringBuffer`) of the `PlugInData` object; if the data field is empty, the data may be stored as binary data in the `documentData` field of the `PlugInData`. If no data is available in either the `data` field or the `binary` field the PlugIn script can read the file in itself by using the filename field of the `PlugInData` which identifies the file being opened. The BeanShell script for the extension of that file (if any) will be run on the contents of that file as it is opened. This allows you to preprocess files in certain ways.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the PluginData object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

For example, to decompress a file with a file extension `.gz` (e.g., job1`.gz`), you would have a BeanShell script in `open/gz.bsh`

If one open script gives the data a new name which itself has an open script, then the data will be sent through that filter in turn. For example, if in the gz file we have opened in the previous paragraph we select a file `h.html`, then the `open/html.bsh` script will be run on that file before the data comes in. When a script changes the filename it should also add a new entry in the `Vector` stored in the `DC_SOURCE` property of the `DocumentData`. The script should insert the OLD filename (not the new!) as the first item in the `Vector` using the Dublin core methods on the `DocumentData` object. In the case of changing the filename a new property should be added so use the following command:

```
docData.setDCProperty(DocumentData.DC_SOURCE, “old_filename”);
```

Note that in general, the expectation is that opening a file will lock that file, and that the lock can only be reset when the file is saved. Temporary files created by open scripts will not be locked. A readonly file can still be opened; however it cannot be saved to the same filename.

The open plug-ins are also the way in which files using different extensions automatically open with the right editing mode (the correct “notation”). The script must create an entry in the hash table `documentData` with the key “document.notation” and one of the following String values (if no notation is set explicitly, the editor defaults to XML):

- xml

- dtd
- sgml
- text
- bat
- c++
- c
- eiffel
- idl
- java
- javascript
- php
- tex
- sql

The same mechanism can also be used to associate extensions with namespaces, XML schemas, DTDs, RELAX schemas, Schematron schemas, CSS stylesheets, an XSL stylesheets suitable for a browser, and an XSL stylesheet producing some kind of index or overview of the document. The script must create entries in the hash table `documentData` with one or more of the following keys:

- "dtd-location" - The location of a DTD
- "sgml-dtd-location" - The location of a SGML DTD
- "css-location" - The location of a CSS stylesheet
- "index-location" - The location of an Index file
- "relax-location" - The location of a RELAX-NG schema
- "schematron-location" - The location of a Schematron schema
- "schema-location" - The location of a W3C XML Schema
- "xsl-location" - The location of an XSL stylesheet
- "catalog-location" - The location of a Catalog file

If the locations came from an XAR file, they will be pointing to the editor's `schemas/` directory.

At the moment there is no explicit mechanism to dispatch according to MIME type. You could write a plug-in script to examine the incoming MIME type or the first part of a document to determine the namespace and generic identifier of the first element, and then select the schema etc. accordingly. But note that if you use this approach on, say, the .xml extension itself, the result is less flexible to maintain than using the extension to determine the document type.

save/

The save plug-ins are checked whenever a file is to be saved. The script takes text in the `data` field (a `StringBuffer`) of the `PlugInData` object. The BeanShell script for the extension of that file (if any) will be run on the contents of that file as it is saved. Before the content of a document is saved to a file the data must be converted to a binary format

stored in the `binary` field of the `PlugInData` object. If this field is set after the PlugIn script has executed this binary data will be saved to the file. If the data is still stored as characters (the `data` field is non `null` and the `binary` field is `null`) then the standard transcoding box will be displayed before the data is saved. This allows you to postprocess files in certain ways.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the PluginData object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

For example, to process a file with a file extension `.xtm` (e.g., `job1.xtm`), you would place a BeanShell script in `save/xtm.bsh`

If there `documentData` hashtable of the PluginData has an entry “dc:source”, then this Vector constains the names of files created by other open plug-ins. the plug-in framework will use (the extensions) of these names to attempt to perform the reverse actions when saving a document.

For example, if `documentData` contains `job1.gz` then `h.html`, then first the file is processed by any plug-in `save/html.bsh`. Then the plug-in for `save/gz.bsh` will be run on the output of that.

Note that if the document was opened from an archive contained in an archive, (e.g. a ZIP file inside a ZIP file), only the outermost archive will have been locked. No special features are provided to support saving to archives within archives, except that the names of the intermediate archives are available to a plug-in in `documentData` field with the key “dc:source”.

close

The close plug-ins are called whenever a file is closed. The Beanshell scrtips for the extension of that file (if any) will be run. This allows scripts to clean up data, post-process data, check data back into version control systems and make new versions of dependent publications.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PlugInData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

openName/

The openname plug-ins are checked whenever a file is to be opened. The BeanShell script for the extension of that file (if any) will be run on the name of that file as it is opened. This allows you to create handlers for rights-checking, file permissions and versioning.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PlugInData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

For example, to notify some access-logging system file that you are editing a file with the extension `.audited`, you would place a BeanShell script in `open/audited.bsh`

Another use of openName would be to check the rights associated with a filename before attempting to access it.

saveName/

The savename plug-ins are checked whenever a file is to be saved. The BeanShell script for the extension of that file (if any) will be run on the name of that file as it is saved. This allows you to create handlers for rights-checking, file permissions and versioning, for example.

All errors or exceptions must be handled by the plug-in. Metadata for the document being edited can be added or found in the `documentData` field passed as part of the `PlugInData` object. Data required between invocations of plugins can be added or found in the `sessionData` field of the options system. (The Options system also allows you to add new categories and panels to the options box, for persistent storage and convenient user storage.)

For example, to automatically update some validity registry upon saving for files with the extension `.xml` , you would place a BeanShell script in `openname/xml.bsh`.

companion/

The companion plug-ins are checked whenever a file is opened. If the opened file has some companion file—a file in the same directory with the same name and a significant extension—then the BeanShell script for that extension will be run on the companion file.

The example companion is the script `companion/odrl.bsh` which informs the user of the Open Digital Rights Languages rights. If I open a file `xxx.xml`, and there is also a file `xxx.odrl` in the same directory, then the script `odrl.bsh` will be run using `xxx.odrl` as its argument. If three letter extension are also used (e.g. `.odr`), then there should also be a script `companion/ord.bsh`.

Another use of companion plug-ins might be if there is a script `companion/make.bsh` which, when I open a file xxx.xml that has a file xxx.make in the same directory, would first run a makefile to update the various subentities of the document.

SERVICES APIS

The Services APIs provide some components and facilities which plug-ins can use. This reduces programming effort for common kinds of plug-ins and allows the plug-ins to keep the same look-and-feel of the *Topologi Markup Editor*. The major component of the Services API is the `Services` object itself which can be accessed through the `PlugInData` object with the `getServices()` method:

```
Services service = pid.getServices();
```

The Services object will give to access to three utility objects that are useful when you want to perform different tasks in the editor. These objects are:

- `FileChooser` - Select files from the filesystem
- `Options` - Access the editor options
- `StatusBar` - Write messages to the user using the status bar in the *Topologi Markup Editor*

FileChooser

The FileChooser object can be used to bring up a dialog window where the user can select files from the filesystem. Three methods can be used to show this dialog:

- showDialog(Component parent, String approveButtonText)
- showOpenDialog(Component parent)
- showSaveDialog(Component parent)

The first method can be used to show a dialog with a customized text on the confirm button. The other two methods will bring up a dialog with the text “Open“ and “Save“ respectively on the confirm button. The `Component` that should be sent to the methods can be set to null if you want the dialog to be separate from the editor application but it’s recomended that this parameter is set to the editor frame itself. This way the FileChooser object is tied to the editor application which prohibits other actions in the editor when the FileChooser is open. The editor frame can be accessed through the Services object as well with a call to the method getReference-Frame(Main.APP_MARKUPEDITOR).

Before showing the FileChooser you can also use the following methods to control the type of selections you can make:

- setFileSelectionMode(JFileChooser.FILES_ONLY);
- setMultiSelectionEnabled(false);

The setFileSelectionMode() method can take the following values as an argument:

- JFileChooser.FILES_ONLY - Only files can be selected
- JFileChooser.DIRECTORIES_ONLY - Only directories can be selected
- JFileChooser.FILES_AND_DIRECTORIES - Both files and directories can be selected.

The setMultiSelectionEnabled() controls whether or not more than one file (or directory) can be selected at the same time. Depending on which option is specified the following methods can be used to access the selected value:

- getSelectedFile() - This method will return a `java.io.File` object
- getSelectedFiles() - This will return an array of `java.io.File` objects (`File[]`)

Here is an example of how the FileChooser object can be user in a PlugIn script:

```
...
    // Set properties so that we only select files and only single selection option
    pid.getServices().getFileChooser().setFileSelectionMode(JFileChooser.FILES_ONLY);
    pid.getServices().getFileChooser().setMultiSelectionEnabled(false);

    // If we don't choose a file the report an error and exit the script
    if (pid.getServices().getFileChooser().showOpenDialog(
       pid.getServices().getReferenceFrame(Main.APP_MARKUPEDITOR)) !=
       JFileChooser.APPROVE_OPTION) {
          pid.getServices().getStatusLine().setMessage("No file was selected.");
          pid.setStatus(PlugInData.CANCELLED);
          handler.handleScriptResult(pid);
          return;
    }

    // Get the selected file
    File file = pid.getServices().getFileChooser().getSelectedFile();
    // Print the absolute path to system out
    System.out.println("The selected file was: " + file.getAbsolutePath());
...
```

To use the FileChooser object and all it's functions in a script the following import statements need to be included in the script:

```
import com.topologi.tme1.io.plugins.*;
import com.topologi.tme1.services.statusbar.*;
import com.topologi.tme1.main.*;
```

Options

The Options API provides access to the options panel data. You can get and set option data, create new options panels, and bring panels to the top. The Options object is accessed through the Services object with the getOptions() method.

The Options API also provides

- a hash table `sessionData` for storing data that will persist for the life of the editor session, with keys such as "`import.ftp.url`" for saving the url given in the last invocation of the plugin that handles FTP import;
- access to `colorScheme` object, so that buttons and tabs can share the same look of the rest of the editor.

A plug-in may access data from the options panels. For example, default system colors (these are used rather than a Java Pluggable look and feel) are available using

```
Color pid.getServices().getOptions().colorScheme.getBackground(); // Background for dialog box
Color pid.getServices().getOptions().colorScheme.getBars(); // for text bars carrying information with same emphasis as tabs
Color pid.getServices().getOptions().colorScheme.getButtons(); // for tabs and major buttons
Color pid.getServices().getOptions().colorScheme.getContents(); // text fields
Color pid.getServices().getOptions().colorScheme.getInsetButtons(); // buttons inside a frame
```

```
Color pid.getServices().getOptions().colorScheme.getOtherButtons(); // buttons that don't call attention to themselves
Color pid.getServices().getOptions().colorScheme.getSelectedButton(); // selection color of tabs and major buttons
```

(A plug-in may add extra categories (pages) to the Options system. This allows data to be stored between sessions, and provides a convenient place for users to set their preferences. However, the facilities for this are not fully-developed for the current version of the Extensions API. In particular, options will not be added at start-up: plug-ins must each make sure that the options they require have been added. Also, it is not straightforaward to create a full options panel as part of a single script.)

StatusBar

The StatusBar API provides access to the message bar at the bottom of the editor's window.

A plug-in may send a message to the message box at the bottom of the editor window.

```
pid.getServices().getStatusBar().setMessage("Error...");
```

Note that executions will continue even if a message has been sent. To terminate execution, either the plug-in set the status of the `PlugInData` object to `PlugInData.CANCELLED` or `PlugInData.ERROR` or it can put some data in the `PlugInData.error` field.

All the above objects that exist in the Services object can also be accessed directly through the PlugInData object for ease of use:

```
pid.getFileChooser();
pid.getOptions();
pid.getStatusBar();
```

Paths

The locations of various paths are available at

```
String p.coptions.get??();
```

The Plug-in API provides no objects to easily access the editor itself or to edit documents directly. This provides a basis for document security, by restricting access to documents to plug-ins run at certain times

Encoding

To work successfully in a multi-lingual and multi-character set world, programmers must be very clear on what they need to do. The *Topologi Markup Editor* provides state-of-the-art character encoding handling in a convenient API.

All documents being read from the outside world should, unless the programmer knows clearly which encoding has been used, be read into a plugin as *"binary"* data. The editor component will, when it recieves only binary data, run the character encoding methods itself, as a last resort.

However, if the writer of a plug-in needs to access the data as characters (i.e. to scan it), then the plug-in itself should first invoke the plug-in handler to make sure the encoding is correct.

Continue to the following section for more information on encoding.

External Processing

It is possible to set an external process that will be used as part of the text encoding process. It is made part of the file open process in order to take advantage of available buffers.

For example, when opening an HTML file, the *Tidy* program can be run to repair an incoming HTML file. To register a process, a script should simply

1. Set a variable to say the input encoding to be for the input file
   ```
   p.getDocumentData().setProperty(
   "run-script-on-transcoded-data-write-encoding",
   "UnicodeLittleUnmarked");
   ```
2. Set a variable to say the encoding to be used for the file with the data being returned
   ```
   p.getDocumentData().setProperty(
   "run-script-on-transcoded-data-read-encoding", "8859_1");
   ```
3. Set the process to be run
   ```
   p.getDocumentData().setProperty(
   "run-script-on-transcoded-data", "command" );
   ```
4. Run the InputTranscoder.

See the code in `plugins/open/topologi/html-02.bsh` (or a more recent version) for an example. The process to be run should expect one file name as its argument: the InputTranscoder will create this temporary file in the systems temporary directory. Your command string does not provide this filename, the InputTranscoder will; the file will be deleted when the transcoder finishes.

Your process should use that file for both input and output: if you are want to use some shell commands that write to a different file than the input file, you should wrap them in commands to do the renaming.

SECURITY

The Topologi Markup Editor provides no special mechanism for security within a work group. Plug-ins, diaries and messages will be interchanged freely. However plug-in scripts and their jar files are not provided with access to documents-in-progress nor to the peer-to-peer objects.

A rogue insider could deploy plug-ins with undesirable properties which monitors documents before and after editing, and uses standard sockets or URLs to communicate.

However, this is no different from any other editor, so it should not be considered a significant security risk.

For downloading plug-ins and updates from Topologi, this option is off by default.

XAR

An XAR file is an XML Application aRchive. It is a simple format to allow the interchange of various metadata used by a document type: schemas, stylesheets and vendor-specific configuration information.

The *Topologi Markup Editor* is trialling the DZIP format while XAR is under discussion. A `.dzp` file can be placed in the appopriate subdirectory of the `xar/` directory and the *Topologi Markup Editor* will treat it as an XAR file: it's name (processed as for the plugins) will appear on a menu on the Document Information box (under the File Menu). Selecting a DZIP archive will unzip it under the `schemas/` directory. Certain well-known extensions are used to locate files: the first */*.dtd will be used as the DTD, and written to the "doctype" entry in the `documentData` field.

See http://www.topologi.com/public/dzip.html. Basically, the first file with a dtd extension in the root of the DZIP file will be used as the DTD; the first file with an xsd extension will be used as the XML Schema; the first file with a .sch extension will be used as the Schematron scheme; etc.

The Sidebar Assistant is configured using an NII (Nested Information Interchange) XML file. This must be placed under `vendor/topologi.com/*-nn.nii`

To see the DTD for an NII file, use the *Sidebar* tab in the *Document Type* tool, and select edit.

The current version of the editor does not provide a specific tool to create XAR files. The best approach is to lay out the files in a dirctory, then use a common ZIP utility to create the DZIP file, correcting the extension as appropriate. You can use the *Edit>Export>Export to ZIP* menu item to add files to a ZIP file, but this may be a little tedious.

DESIGN OVERVIEW

FIGURE 1. Logical Components

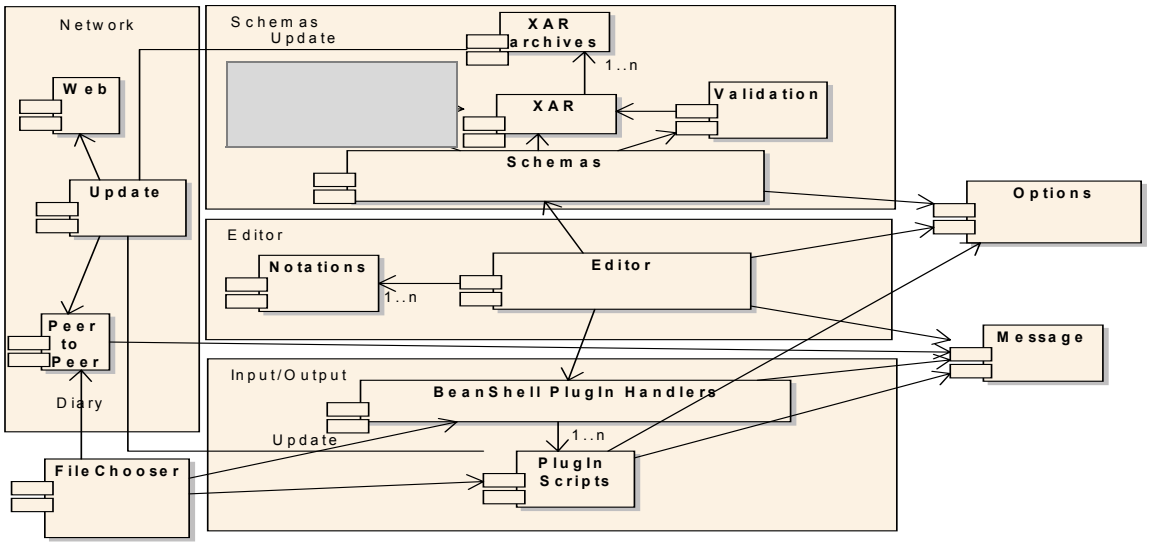


FIGURE 2. Library Components