# APT Tool Development Guidelines

## 1. Overview

APT is designed to be as general purpose as possible. To achieve this the architecture is broken down into three distinct, packages: Tina, APT, and HST APT, from most generic to most specific.

The following documentation conventions are used in this document:

- ***classname: bold, italicized***
- *method(): italicized*
- `code examples: courier font`

The online API documentation for APT is packaged with everything else at:

http://ra.stsci.edu/apst/apt/documents/develop-API/

## 2. Layered Framework

The architecture of Tina, and therefore of APT and the HST implementation of APT uses four layers. Each layer is built upon, and dependant upon the layers below it. The first layer is the persistent storage for proposals which is currently XML files on the local file system. The second layer is the APT data model as a domain specific extension of the Tina data model. The data model understands and uses and the underlying storage layer.

The third layer is the controller, which depends upon the Tina data model of which the HST proposal data model is an implementation. APT tools communicate with the controller, and make direct use of the data model.

The fourth layer is the GUI or view which presents an interface for interacting with the controller and data model. APT tools provide GUI components that are plugged into the top level browser.

## 3. APT Tools

Tools for APT are required to implement the ***TinaToolController*** interface, or to extend the ***AbstractTinaToolController*** class which provides default implementations of most of the interface methods.

### Tina/APT Tool Interface

The ***TinaToolController*** interface provides methods that describe the tool for the Tina framework, allow that framework to pass the Tina Context to the tool, and allow the controller to obtain GUI views from the tool to place into the browser and preference windows. Note that preference panels will be separate the tool view, but the API for obtaining these panels is not yet in place.
The tool describes itself to Tina via the methods:

| | |
|---|---|
| *getToolName()*: | Returns the full name of the tool to be used in the tools menu. |
| *getToolShortName()*: | Returns a shortened name to be used in the tool bar. |
| *getToolTipText()*: | Returns a longer string describing the tool more fully. This will be used as the tooltip for the tool. |
| *getToolIcon()*: | Returns an icon that may be used in either the menu or the tool bar. |

The ***TinaController*** calls the following methods on the ***TinaToolController:***

| | |
|---|---|
| *activate()*: | Called when the tool is made active and its view is made visible by placing it into the browser. |
| *deactivate()*: | Called when the tool is made inactive and its view is hidden by removing it from the browser. |
| *setContext(**TinaContext**)*: | When the tool is first loaded into the Tina framework it will be passed the context via this method. |
| *getNewView(int):* | Should return a GUI component to place into the browser. As described below, additional calls should generally return a new view, not the same one. The single parameter indicates the expected placement of the view in the GUI. The parameter should be either: **javax.swing.SwingConstants.TOP** or **javax.swing.SwingConstants.BOTTOM**. |
| *getToolMenus()* | Should return an array of JMenu objects to be placed in the menu bar when this tool is active. |
| *getToolButtons()* | Should return an array of JButton objects to be placed in the tool bar when this tool is active. |

## <u>Tina Context</u>

A central component of the Tina system is the context, a single instance of ***TinaContext***. The primary role of the context is to convey user object selection to the tools. In addition, the context maintains other attributes of the selection which may be user or tool defined:

| | |
|---|---|
| • user selection | The set of objects selected by the user in the hierarchical editor. |
| • inferred selection | Tool specific selection of additional objects. Any tool can set the inferred selection, and tools may decide whether or not they react to the inferred selection. The hierarchical editor will set the inferred selection to the objects selected by the user, and the objects contained by the user selection. The spreadsheet editor will listen to the inferred selection and display it. |
| • lead selection | The primary or lead selection. This is the primary selection within the overall group of objects that is selected. Any tool can set this attribute, and any tool can read it. The hierarchical and spreadsheet editors will set this to the most recently selected object. The spreadsheet editor will select this object unless the selection is set within another view of the spreadsheet editor itself. |
| • proposal selection | The currently selected proposal. Saving, and undo and redo are tied to the current proposal. |
| • everything | All of the TinaDocumentElements in all of the loaded documents |

## Context Interface

Methods are provided to access the objects in the context, and in each case except the user selection, methods are provided to set the elements of the context. See the API for the relevant methods at:

http://ra.stsci.edu/apst/apt/documents/develop-API/edu/stsci/tina/controller/TinaContext.html

## Listener Interfaces

There are four listener interfaces that tools may optionally implement:

***TinaContextListener***:

A tool will generally implement the ***TinaContextListener*** interface, and when given the ***TinaContext*** the tool should register itself as a listener of that context. Note that if the tools extends ***AbstractTinaToolController***, then the default *setContext()* method will register the tool as a listener for whichever interfaces the tool implements.

This interface defines a single method that is called whenever the context changes:

| | |
|---|---|
| * *contextChanged()* | Notification that the context has changed. As there is only |

one instance of the context, no event is used. The tool must query the context to discover what has changed.

***TinaInferredContextListener***:

* *inferredContextChanged()* Notification that the inferred context has changed.

***TinaCurrentDocumentListener***:

* *currentDocumentChanged()* Notification that the current document has changed.

***TinaQuitListener***:

* *tinaWillQuit()*: Called just before the system quits allowing tools to clean up before exiting.

**Tool activation and deactivation**

A requirement of the APT system is that tools in the background not impact the responsiveness of the foreground tools and browser. We have decided to use the java event model to support data model and context change notification. This means that the main system thread will be running the *contextChanged()* methods in every tool when these changes occur. Since we have adopted this default event model we must meet the preformance requirement by convention, ensuring that the tools return quickly from these callbacks.

This places the burden on the tools to consume minimal CPU in the background, which means that they need to know whether they are activated or deactivated. This notification is communicated through the *activate()* and *deactivate()* methods. When a tool is activated it should make itself consistent with the current context, and with the state of the objects in the context. When it is deactivated, it may need to defer responding to data model and context changes.

If a tool is very fast then it may be able to essentially ignore the activation and deactivation. This may also apply when the tool requires a specific use action, such as a button push, to 'run'. Other tools may need to go to more effort to defer processing.

## 4. APT Tool Design Pattern

**Intent:** To ensure separation of views from tool controls and models thereby supporting multiple views into a tool.

**Motivation:** Flexibility. We have already found that having two instances of the

Spreadsheet Editor open simultaneously has some value, however we do not want two full implementations of the tools at once. Rather we would like to have multiple views into a single tool for performance and memory use reasons. We believe that there will be more instances in which it is desirable to have two or more views and this design pattern separates the computational logic and state of the tool from the GUI display associated with the tool.

**Structure:** The pattern has three elements: an implementation of the TinaToolController interface, a GUI component that provides a view of the tool, and a listener interface through which multiple instances of the view are notified when the tool changes. The view implements the listener interface and each time a new view is created it is registered with the tool controller as a listener. This pattern was used in the implementation of the current set of built-in tools.

The tool controller interface may be implemented by the core tool itself, or may by a wrapper around the tool. In either case the tool should defer full initialization until the first time it is activated via the activate method. The Tina framework will not call getComponent until after the first activation of the tool, but the tool controller will be instantiated at APT startup time.

This pattern supports the following:

* startup of the tool without full initialization
* notification when initialization is necessary
* separation of the tool from the GUI
* multiple simultaneous views of a single tool

UML diagrams for a particular implementation of this pattern may be found at:

http://ra.stsci.edu/apst/apt/apt-framework/DesignReview-20010717/APT-Architecture.ps

## 5. APT Data Model

The Data Model is accessed though the ***TinaContext*** which may contain any number of elements from the currently loaded proposals.

We recommend that tool developers consider defining an interface for the data model objects that their tool is to work with. While this is not required, we think it is a good idea for the following reasons:

* The tool can select objects from the context based upon type (i.e. those that implement this interface.

* decouple the data model from the tools. This will make it much easier to add a different data model, that the existing tools can work with without modification.

For example, the Visit Planner has requested that the APT data model implement a ***edu.stsci.apt.model.SchedulingUnit*** interface.

The interfaces are defined in the APT/Model repository in CVS, and implemented by the APT and HST APT data models. The APT developers will help with implementation of such an interface if one is defined. In developing a tool the developer will need to checkout just the APT data model in addition to their tool.

**Tool Data**

Tools may associate arbitrary data with objects in the data model to be stored and reloaded with the proposal. The TinaDocumentElement interface supports the methods:

| | |
|---|---|
| *putToolData(String iKey, JdomBinding iObject):* | which is used to place a data element into the object. |
| *getToolData(String iKey)*: | which is used to recover that data from the object. |

Implementation of the ***edu.stsci.tina.model.JdomBinding*** interface allows the data to be written to the proposal file, and then reconstructed when the proposal is loaded. Implementation of this interface will require that the jdom.ajt library be included in your project.

In order to enable Tina to reconstruct the object correctly it needs to know what class the root Jdom Element will be handled by. This is accomplished by registering the class with the static ***edu.stsci.tina.controller.JdomBindingFactory*** via the follwing method:

*JDomBindingFactory.addJdomBinding(String iTag, Class iClass):* which registers the the given Element name as handled by an instance of the given class.

This registration must occur during startup of APT (during loading of the tool) to ensure that the class is registered before a proposal is loaded.

**NOTE:-** The data model is still in flux, so don't build too much dependancy upon it yet. If you define an interface for the data model to implement, this instability should be much less of an issue as we are not likely to your interface.

## 6. Undo Support

APT maintains a single global undo stck of unlimited depth. This stack is not, however, persistent across separate runs of the tool. The data model itself deals with most of the undo management as in general the only changes that are undoable are those to the data model itself. The data model posts undo events in response to all changes it receives.

In general then, APT tools will not need to deal with undo/redo as it will be handled automatically, however there are certain scenarios where the default behaviour is insufficient.

1. A number of changes are being made to the data model but they should all be considered a single operation for undo/redo purposes.

2. A change in the tool itself which does not change the data model should be undoable.

Undo is handled through an instance of **UndoableEditSupport** which may be obtained from the TinaController via the method *`getUndoSupport()`*. Through this object a tool can handle the two situations described above.

1. Changes may be aggregated into a single undo/redo event through the use of two methods on the **UndoableEditSupport**. Before making any of the changes call *beginUpdate()*. After all changes have been made call *endUpdate()*. All undo events posted between these two calls will be aggregated into a single undo event.

2. A tool may post its own undo events by calling *postEdit(UndoableEditEvent e)*.

## 7. Cut/Copy/Paste

Cut/Copy/Paste is a little more cumbersome for individual tools. Management of the clipboard and associated actions has been bundled into a single class called **TinaEditSupport**. The run time instance of this class may be obtained through the **TinaController** via the method *getEditSupport()*.

To receive call-backs from Tina when these global actions are invoked a TinaEditListener must be implemented and registered with the support object. Note that only one listener may be registered at a time as that listener is considered the owner. The owner is the only one that can enable and disable the actions, and the owner is the one that will be notified when an action is invoked. As a result it is up to the tool itself to figure out when it should grab ownership via *setOwner(TinaEditListener l)* or even when it should specifically relinquish ownership via *removeOwner(TinaEditListener l)*. It is not normally necessary to relinquish ownership explicitly as another tool may grab ownership at any time. Ownership should be based upon focus. Good luck with this :-)

There are five actions associated with this object (Cut, Copy, Paste, Duplicate, Delete) and two clipboards. The actions may be indivually enable or disabled by the owner, or they be enabled or disabled as a group. Any or all of the actions may be implemented by the TinaEditListener.

There are currently two clipboards in the TinaEditSupport. A system clipboard which supports transfer outside of APT and a purely internal clipboard. You can grab the system clipboard to interact with via *getSystemClipboard()*. The internal clipboard currently acceptas and returns a Vector of objects (probably less than ideal but easy to work with). Objects may be placed on the internal clipboard via *setObjectClipboardContents(Vector v)* and they be accessed via *getObjectClipboardContents().*

## 8. Source Code

All APT source code is managed in a CVS repository on ra.stsci.edu. CVS can be configured to access this repository remotely (if you have an account on ra) using a CVSROOT of :ext:ra.stsci.edu:/usr/ra/cvsroot.

APT is packaged as a library and is placed in the APT/Libraries directory with the other files upon which it depends. To run APT all of the jar files in this directory will need to be in the classpath.

## 9. Tool Packaging

For delivery tools should be packaged as a Jar file to be placed into the APT Plug-Ins directory. In addition, a special attribute must be set in the manifest file. The class or classes in the Jar file that are APT tools should have a TinaTool attribute with value true. For example, the TextTool contains this entry in the manifest:

```
Name: edu.stsci.apt.TextTool
TinaTool: true
```

To create a Jar file with the necessary entries in the manifest you will create a text file containing the manifest information and reference this file when creating the Jar. For example:

```
jar cmfv ../../APT/TestTool/edu/stsci/apt/TextTool.MF TextTool.jar edu
```

adds the information in ../../APT/TestTool/edu/stsci/apt/TextTool.MF to the manifest in the new TextTool.jar file.

Resources for the tool should generally be packaged with the tool in its Jar archive. In order to access such resources you **must** use the ClassLoader associated with the tool rather than the system ClassLoader, as the system ClassLoader is not used to load the tool, e.g:

```
  getClass().getResource("/images/ToolIcon.gif");
```

rather than:

```
  ClassLoader.getSystemResource("/images/ToolIcon.gif");
```

The later will work when the tool is the starting point for running APT during development, but won't work when the tool is loaded directly by APT as a plugin.

Tools and Data Models are both plug-ins to Tina/APT and Tina must be able to find them in order to load them. This can be achieved in one of two ways. The default mechanism is to place the plug-ins in a folder called "Plug-Ins" directly below the application start-up folder. Alternatively, the folder containing the plug-ins can be specified through the system property: tina.plugins.

The delivered plugins are in the cvs repository in the module APT/Plug-Ins. Note that the HstDocumentModel.jar plugin is necessary for APT to function. If it is not loaded as a plugin then it should be included in the classpath and an explicit call made to:

```
     addDocumentModel(HstProposalSpecification.class);
```

For development this packaging of a tool is unnecessary, and even inconvenient. For development we recommend that the tool itself be used as the starting point with a *main()* which first instantiates the ***edu.stsci.hst.apt.controller.HstAptController*** and then uses the *addTool(**TinaToolController**)* method to add an instance of the tool to APT, e.g:

```
  public static void main(String[] args) {
     HstAPTController lController = new HstAPTController();
     lController.addTool(new TextTool());
  }
```

This approach allows APT to continue to load the default set of plugins. Alternatively, the class ***HstAptController*** may be extended and used as the starting point. In this case the public method *loadPlugIns()* should be overriden to add the tool, e.g:

```
  public static void loadPlugIns() {
     addTool(new TextTool());
     addDocumentModel(HstProposalSpecification.class);
  }
```

By overriding this method you prevent the default set of plugins from loading which may or may not be desirable.

**To see the TextTool example, checkout APT/TextTool from the CVS repository. A more complex example tool is the TinaSpreadsheet which is part of Tina itself.**