Creating Zenoss ZenPacks

September 2009

Jane Curry

Skills 1st Ltd

www.skills-1st.co.uk

Jane Curry
Skills 1st Ltd
2 Cedar Chase
Taplow
Maidenhead
SL6 0EU
01628 782565

jane.curry@skills-1st.co.uk

 © Skills 1st Ltd

Synopsis

ZenPacks are the extension mechanism provided by Zenoss to build new functionality and also to easily port customisation from one Zenoss server to another. Some documentation is provided in the Zenoss Developer's Guide 2.4; this paper is intended to enhance and extend that documentation, including a sample ZenPack.

The process of creating, modifying and exporting ZenPacks is discussed, along with debugging hints. The sample ZenPack explores:

- creating new object classes and relationships
- creating new collector modeler plugins to populate the new classes with data
- creating skins to display web pages for the new types of object
- creating performance data templates for the object classes.

It is assumed that the reader is familiar with basic SNMP concepts and with standard Zenoss configuration techniques.

This paper was written based on a stack-built Zenoss Core 2.4.1 on SuSE 10.3. The hostname of the Zenoss server is *zen241.class.example.org.*

Notations

Throughout this paper, text to by typed, file names and menu options to be selected, are highlighted by *italics;* important points to take note of are shown in **bold.**

Table of Contents

1 What are ZenPacks?....4
2 The process of building a ZenPack....5
2.1 ZenPack creation....5
2.2 Exporting and installing ZenPacks....7
3 “Simple” ZenPacks....8
4 Designing complex ZenPacks....10
4.1 Basic principles....10
4.1.1 Configuration data and performance data....10
4.1.2 The Zope Object Database (ZODB)....14
4.1.3 Coding techniques and terminology....15
4.1.4 Databases, Daemons and Directories....23
4.2 Requirements for the sample ZenPack....26
4.3 Creating the sample ZenPack....30
4.3.1 Elements required and their names....30
4.3.2 SNMP data required....32
4.3.3 Creating the ZenPack....35
4.3.4 Adding elements to the ZenPack using Development mode....36
4.3.5 Creating the object class files....38
4.3.6 Creating the modeler plugin files....46
4.3.7 Creating the skins files....57
4.3.8 Linking development mode elements with source mode elements....68
5 Gathering Performance Data....70
5.1 Performance templates for devices....71
5.2 Performance templates for contained devices....74
6 Testing and debugging ZenPacks....77
6.1 Testing....77
6.1.1 Testing new object class files....77
6.1.2 Testing modeler plugins....78
6.1.3 Testing skins files....81
6.1.4 Debugging problems with performance data....83
6.1.5 General testing and debugging hints and tips....85
7 Conclusions....86
References....87
Acknowledgements....88

1 What are ZenPacks?

ZenPacks are the method of extending the standard Zenoss functionality. There are four different sources of ZenPacks:

- Zenoss Core ZenPacks that can be downloaded from http://www.zenoss.com/community/projects/zenpacks/ . These are developed and maintained by Zenoss and are available to both Zenoss Core and Zenoss Enterprise users. They include monitoring of Apache, Dell, FTP, HTTP, LDAP, JMX and MySQL, amongst others.
- Zenoss community ZenPacks, also from http://www.zenoss.com/community/projects/zenpacks/ . These are ZenPacks developed by individuals or organisations and made freely available to the Zenoss community. No support should be implied for them. There are getting to be a large number of community ZenPacks covering the monitoring of VMware, wireless devices, Cisco devices, various switches, printers and several ZenPacks to enhance the reporting of Zenoss devices, events and thresholding.
- Zenoss Enterprise ZenPacks are available at no extra charge to Zenoss Enterprise (ie. paying) customers. They include enhanced VMware and Windows monitoring, fine-grained user management, distributed monitoring and high availability, and a global dashboard, as well as enhanced monitoring of many third-party devices and software packages.
- Write your own ZenPack – and optionally make it available as a community ZenPack

Since Zenoss 2.2, ZenPacks are packaged as Python Eggs. Earlier zip format ZenPacks can be converted to Eggs (see the ZenPacks wiki site at http://community.zenoss.org/trac-zenpacks/wiki/MigratingZenPacks). This packaging is performed automatically for you and you don't need to get into the details of Eggs.

Some of the core and community ZenPacks come with their own documentation; sometimes it is a little sparse. Searching the Zenoss forums is a good way to glean information (http://forums.zenoss.com/index.php).

ZenPacks may be used for two main reasons:

- Creating new monitoring of new types of devices
- Porting either standard or ZenPack configuration of Zenoss, from one Zenoss server to another

Many of the standard Zenoss Graphical User Interface (GUI) menus have an *Add to ZenPack* option; thus action rules, event classes, event commands, user commands, service classes, data sources, graphs, performance templates, reports,model extensions, and product definitions can be simply added to a ZenPack using the GUI (a **simple** ZenPack).

A ZenPack can also add daemons, new device types and user interface features such as menus but this requires programming effort (a **complex** ZenPack). Check Chapter 13 of the Zenoss 2.4 Administration Guide for a short introduction to ZenPacks.

2 The process of building a ZenPack

Before diving into the complexities of writing Python code for complex ZenPacks, step back and examine the **process** that is required. The first question is whether this will be a simple ZenPack that can be entirely created from the GUI, or whether code needs to be written. Either way, the process for creating the ZenPack is exactly the same.

2.1 ZenPack creation

As a Zenoss user with the *Manager* role, use the left-hand *Settings* menu from the GUI and choose the ZenPacks tab.

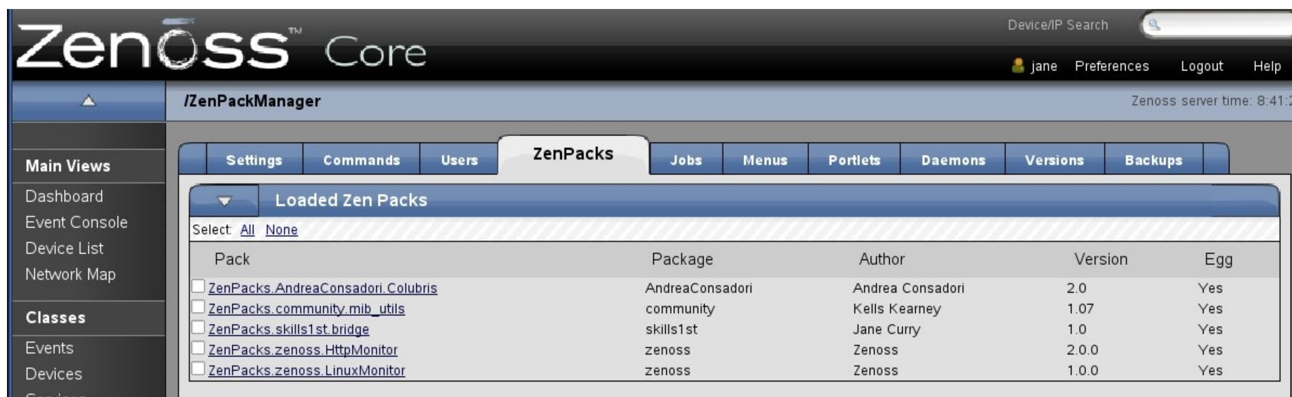

Figure 1: ZenPacks tab from the Settings menu

The dropdown menu options then include:

- Create a ZenPack
- Install ZenPack
- Delete ZenPack

When creating a new ZenPack, the first thing you are asked for is the ZenPack name. ZenPack names are a sequence of three or more package names separated by periods. The first part of the name is always **ZenPacks**. The second part usually identifies the person or organization responsible for the ZenPack. The last part of the name usually identifies the function of the ZenPack (see the screenshot above for examples). Once named, you can then specify other parameters for your ZenPack, like Zenoss version dependency or other co-requisite ZenPacks. You should also specify an author and a version for this ZenPack.

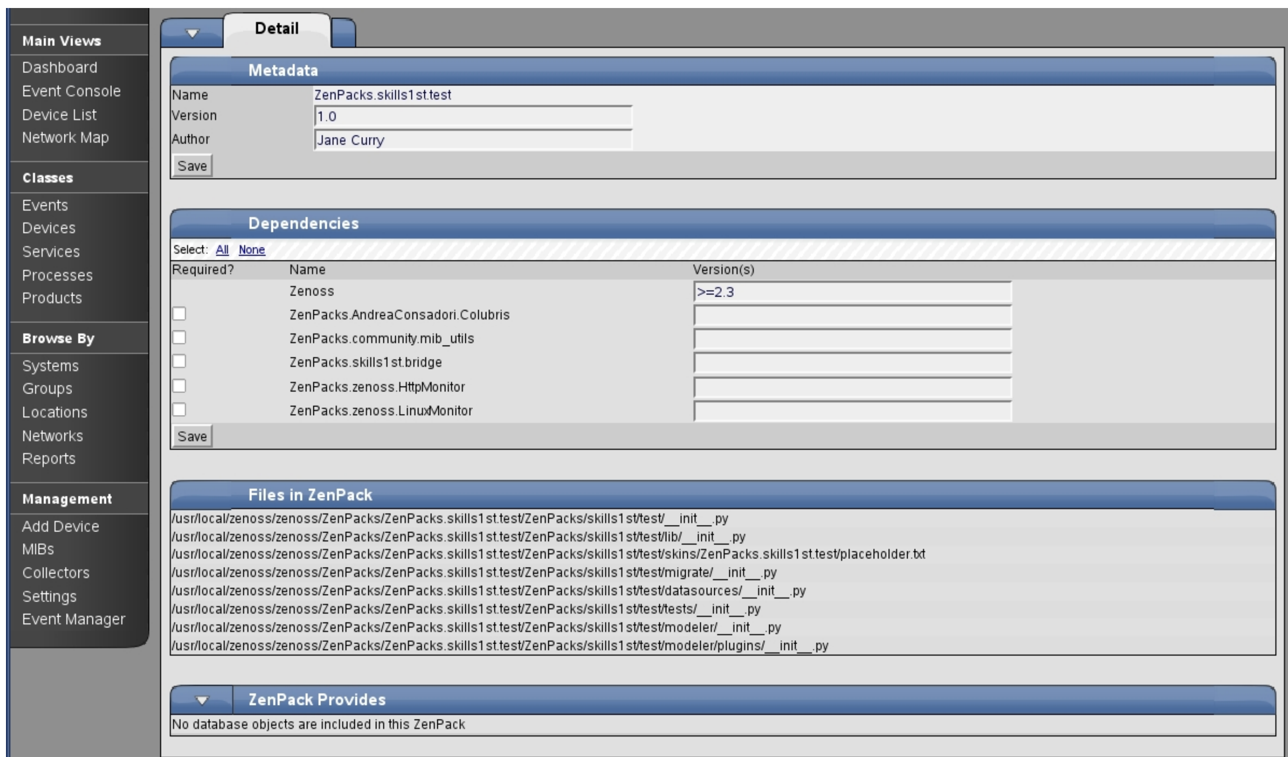

Figure 2: Creation details for a ZenPack

When you create the ZenPack, a directory hierarchy is created under $ZENHOME/ZenPacks as can be seen in Figure 2 above (note that older style ZenPacks used $ZENHOME/Products as the base directory). Each of the directories will have a largely-empty __init__.py file that needs to be there but you should not need to modify it.

The main directory areas that will be discussed in this paper, for the ZenPack called ZenPacks.skills1st.bridge, are:

- ZenPacks.skills1st.bridge – the base ZenPack directory. It contains object class definition files
- ZenPacks.skills1st.bridge/modeler/plugins - modeler plugins for object classes
- ZenPacks.skills1st.bridge/skins/ZenPacks.skills1st.bridge - contains skins files describing web pages associated with displaying aspects of the new object classes

Some of these are rather long-winded but they are created automatically and that is what we have to go with! Once the structure is created, "things" can be added to the ZenPack either from the GUI using *Add to ZenPack* menu options (this is known as **development mode**), or programmatically by placing files in the appropriate directories (**source mode**); indeed, both these methods can be used at any stage.

2.2 Exporting and installing ZenPacks

When you are ready to test the ZenPack on a different system it needs to be exported to create the Python Egg file. Note that the export process also creates the *objects/objects.xml* file - more of this later. From the *Detail* page of the ZenPack, use the dropdown menu to select *Export ZenPack*.

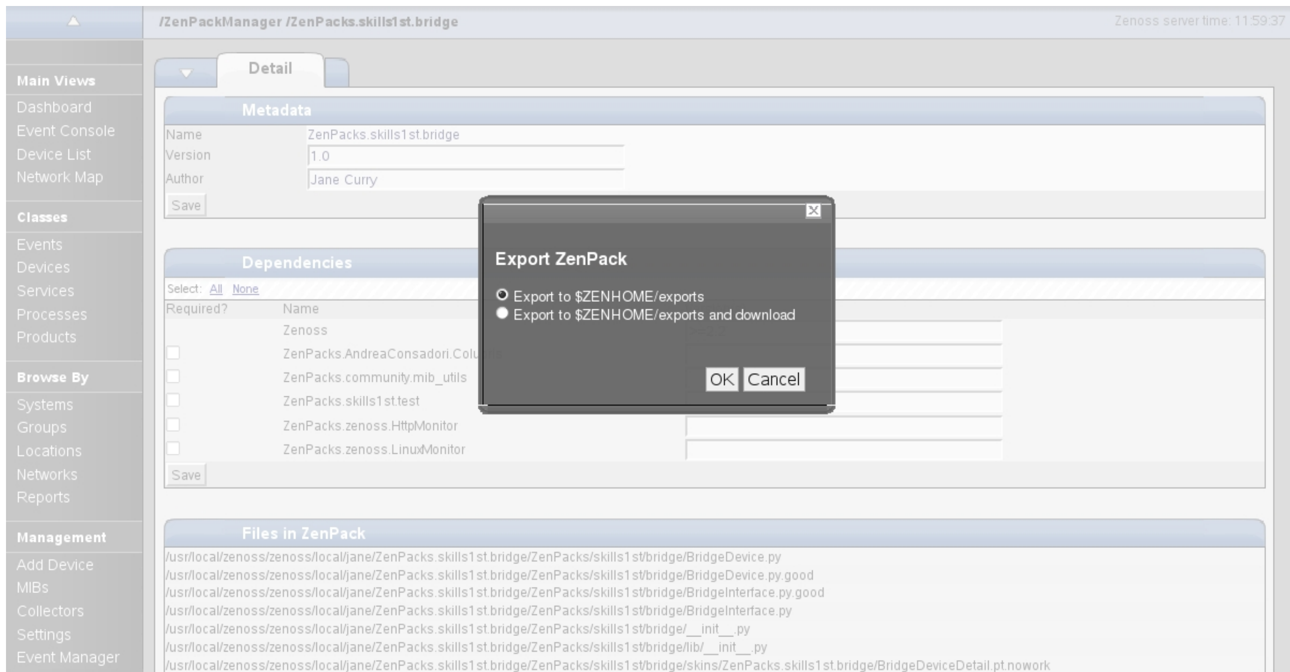

Figure 3: Export ZenPack dialogue

Typically you leave the top radio button selected to just create the ZenPack Egg file in $ZENHOME/exports. The file is first created in your ZenPack's **dist** directory then copied to the $ZENHOME/export directory.

This file can now be moved to the different Zenoss server (perhaps you have a test and a production server?) and installed as any other ZenPack, either using the *Settings* menu *ZenPacks* tab and then the dropdown menu *Install ZenPack*; or you can use the command line:

```
zenpack --install ZenPacks.skills1st.bridge-1.0-py2.4.egg
```

Note the syntax here is 2 hyphens preceding the *install*.

The formal documentation varies somewhat as to what daemons you need to recycle after importing a ZenPack. *zenoss restart* would always be safe but bounces **all** of the daemons. I believe that *zenhub restart* and *zopectl restart* is sufficient. Note that if you forget to recycle the daemons, you may well get error messages from the ZenPacks page and from ZenPack functionality.

When you install an Egg ZenPack, you usually don't have the ability to modify it, thought is possible to do so – see Chapter3, page 22 of the Zenoss 2.4 Developer's Guide for instructions.

If you wish to continue to develop the ZenPack on the new system, the other alternative is to copy the whole ZenPack directory structure and then install the ZenPack with a --*link* parameter (2 hyphens again). This is good practise during initial development as well as in the scenario where you wish to export a ZenPack and then continue to modify it on a different system, largely because if you accidentally use the *Remove ZenPack* menu, it deletes **all** files relating to that ZenPack under $ZENHOME/ZenPacks and this will include any development code you have created if it is stored there.

The sample ZenPack discussed in this paper was created as described above and then moved out of the $ZENHOME/ZenPacks directory using:

```
cp -r $ZENHOME/ZenPacks/ZenPacks.skills1st.bridge $ZENHOME/local/jane
zenpack --link --install $ZENHOME/local/jane/ZenPacks.skills1st.bridge
```

It is perfectly acceptable to reinstall a ZenPack that already exists – it will simply give a warning message that the ZenPack is already installed, but it will do the install. Remember to restart zenhub and zopectl.

The result of the --link parameter is to replace the *ZenPacks.skills1st.bridge* directory hierarchy in the standard $ZENHOME/ZenPacks directory with a single file, *ZenPacks.skills1st.bridge.egg-link*, which simply contains the base directory of where your ZenPack really is. Now, if anyone removes this ZenPack, the only thing that is deleted from $ZENHOME/ZenPacks is this link file, not all your ZenPack code.

From this point, you can continue to develop the ZenPack, either in Development mode, or by writing code in appropriate directories; a mixture of both is perfectly acceptable and all changes will follow this link to actually update code in your private directory.

Zenoss requires a Python module called *setuptools* to create and install eggs. The setuptools module is installed by the Zenoss installer in the $ZENHOME/lib/python directory. Zenoss also provides a module named *zenpacksupport* which extends setuptools . The zenpacksupport class defines additional metadata that is written to and read from ZenPack eggs. This metadata is provided through additional options passed to the *setup()* call in a ZenPack's *setup.py* file.

3 “Simple” ZenPacks

Some ZenPacks can simply be created using the Zenoss GUI; this is especially useful for moving standard configurations from one Zenoss server to another but may also be appropriate when creating ZenPacks to share with other people.

The ZenPack is created exactly as described in chapter 2 above. To add “things” to the ZenPack, simply use the *Add to ZenPack* option that is available on many of the dropdown menus. The following can be added from menus (ie. in development mode):

- Device Classes
- Event Classes
- Event Mappings
- User Commands
- Event Commands
- MIBs
- Service Classes
- Device Organizers
- Performance Templates

You will be prompted as to which ZenPack you wish to add the item to. Objects can be removed from the ZenPack by selecting the checkboxes next to them and using the *Delete from ZenPack* menu item. Devices themselves are the conspicuous omission from this list. Any individual device is usually specific to a particular site and therefore not likely to be useful to other Zenoss users.

To see what a ZenPack contains, simply use the *ZenPacks* tab from the *Settings* menu and choose the appropriate ZenPack.

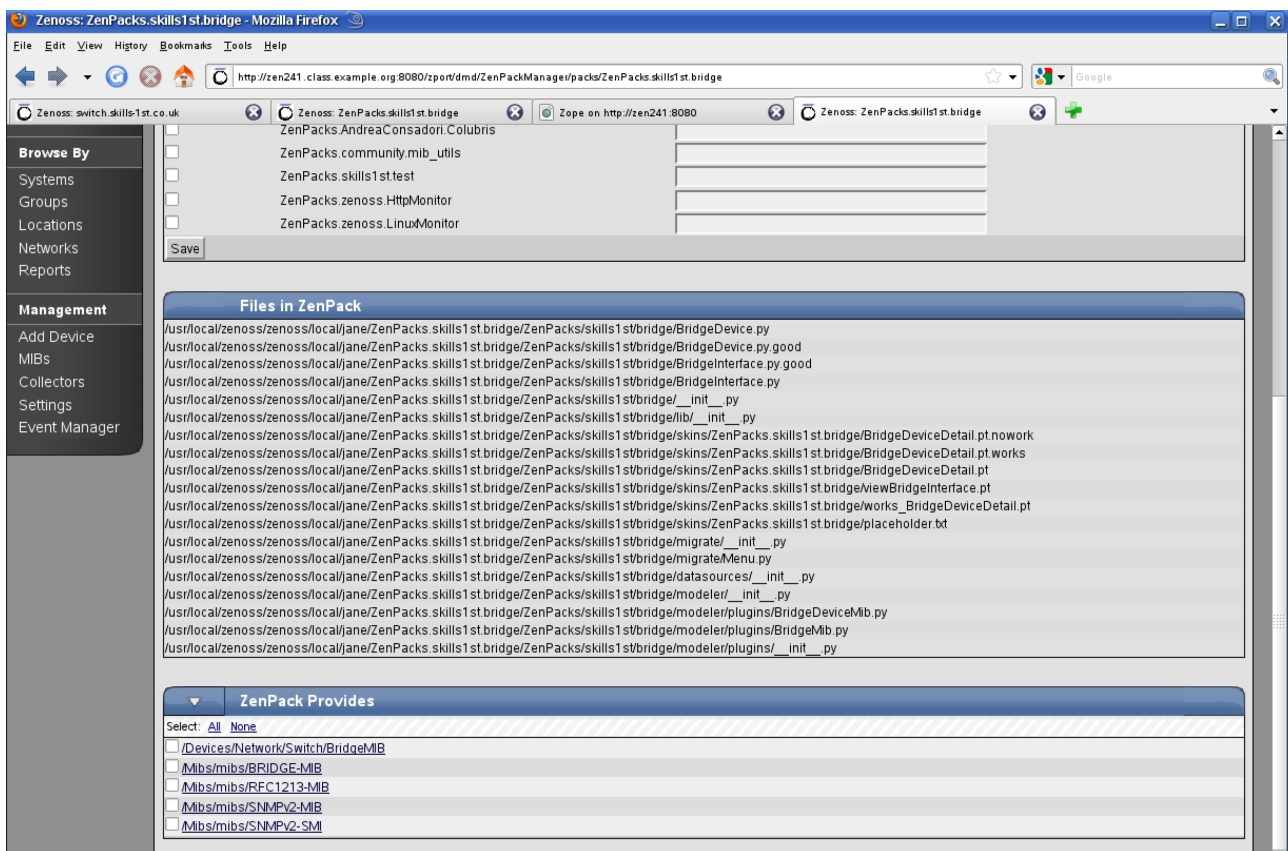

Figure 4: The contents of the ZenPacks.skills1st.bridge ZenPack

When a ZenPack is **exported** (using the dropdown menu from the Detail page of the ZenPack), not only is the Egg file created but it is at this time that all the objects under the “ZenPack Provides” list in the figure above, are written to the objects.xml file under the objects directory of the ZenPack. This file can be inspected with an editor – as the name suggests, it is in xml format.

4 Designing complex ZenPacks

When developing new functionality for Zenoss with ZenPacks, some tasks require more than the standard customisation tools can capture using development mode. For example:

- Supporting a different type of device with different attributes eg. a switch that supports the Bridge MIB
- Polling for SNMP variables from the Bridge MIB to populate these new attributes, such as Port number and the MAC address of the remote device connected to that port
- Displaying web pages that show information about the new device types and their attributes
- Creating new daemons to gather either configuration polling information (modeling) or performance data. Data collection methods for SNMP and ssh are provided as standard but you may need JMX, HTTP or any other method (there are Core ZenPacks that support JMX and HTTP).

This paper will examine the first three of these in detail.

4.1 Basic principles

Before discussing the sample ZenPack requirements and its implementation, let's get some basic principles straight first.

4.1.1 Configuration data and performance data

Zenoss documentation is apt to be a little imprecise sometimes in its terminology and to use different words to mean the same thing. There are two very different concepts to do with collecting data. **Configuration** data is typically polled for every 12 hours and is held in the Zope Object Database (ZODB). **Performance** data is typically polled for every 5 minutes and is held in Round Robin Database (RRD) files from where it can be graphed. The two are very different.

Configuration data is polled for by the **zenmodeler** daemon, using **modeler plugins**. Lots of these are provided as standard with Zenoss under $ZENHOME/Products/DataCollector/plugins/zenoss with separate subdirectories for:

- cmd
- nmap
- portscan
- python
- snmp

Don't be fooled by the directory path containing "DataCollector" - these are configuration modeler plugins used by the zenmodeler daemon and nothing to do with the collection of performance data that typically is collected by the zenperfsnmp or zencommand daemons.

Any device or device class can have several modeler plugins assigned to it. This is configured (rather confusingly) from the dropdown menu and selecting *More -> Collector Plugins*. If this menu was renamed to *Modeler Plugins* then it might be less confusing!

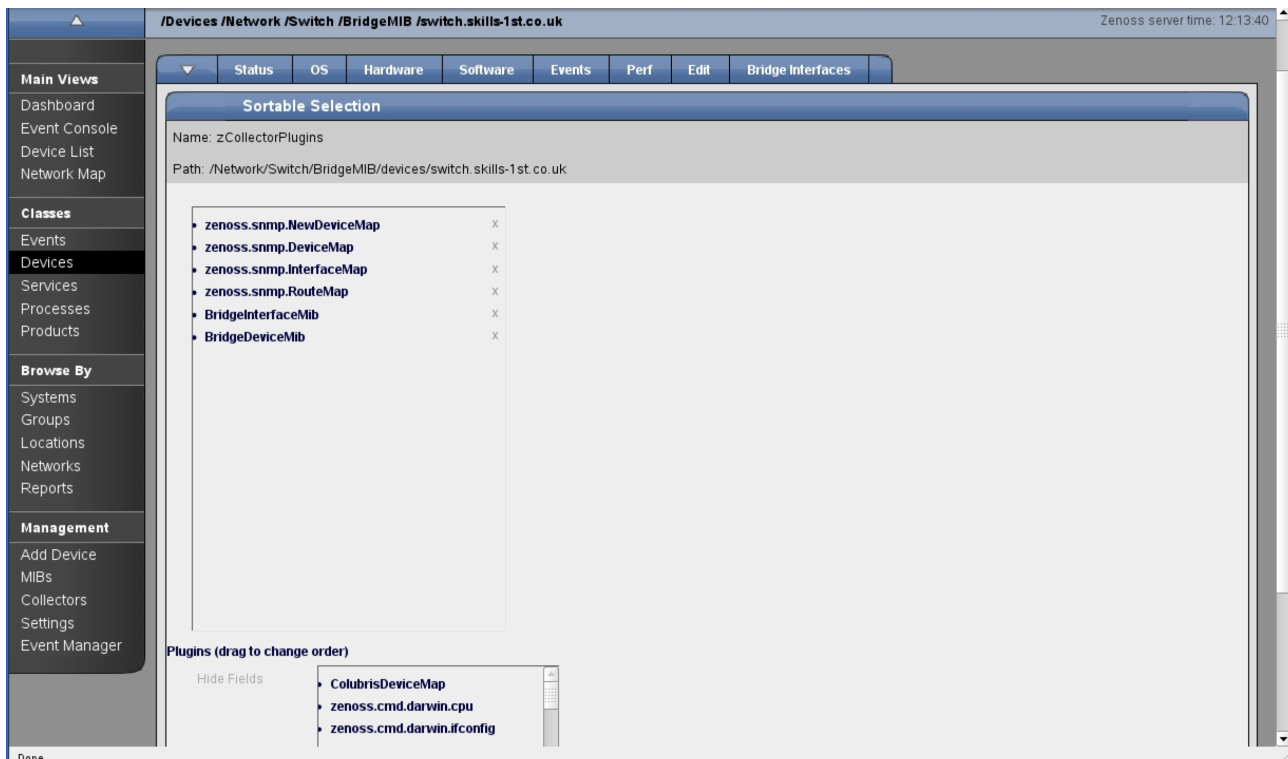

Figure 5: The Collector Plugins dialogue from the dropdown More menu

Initially the dialogue shows the modeler plugins that are currently assigned at the top, and the bottom of the dialogue has “Add Fields”, greyed out (note that prior to Zenoss 2.4, this “Add Fields” was to the right of assigned plugins, rather than beneath them). Although the option appears to be greyed out, click it to see the other modeler plugins that exist. They can be selected simply by dragging them to the assigned area; the order the plugins are run can be changed by dragging the plugins to the appropriate order. Don't forget to use the *Save* button.

Another way to achieve exactly the same effect is to go to the device class or individual device's zProperties page and click on the *Edit* button beside *zCollectorPlugins*, which takes you to exactly the same dialogue shown in Figure 5.

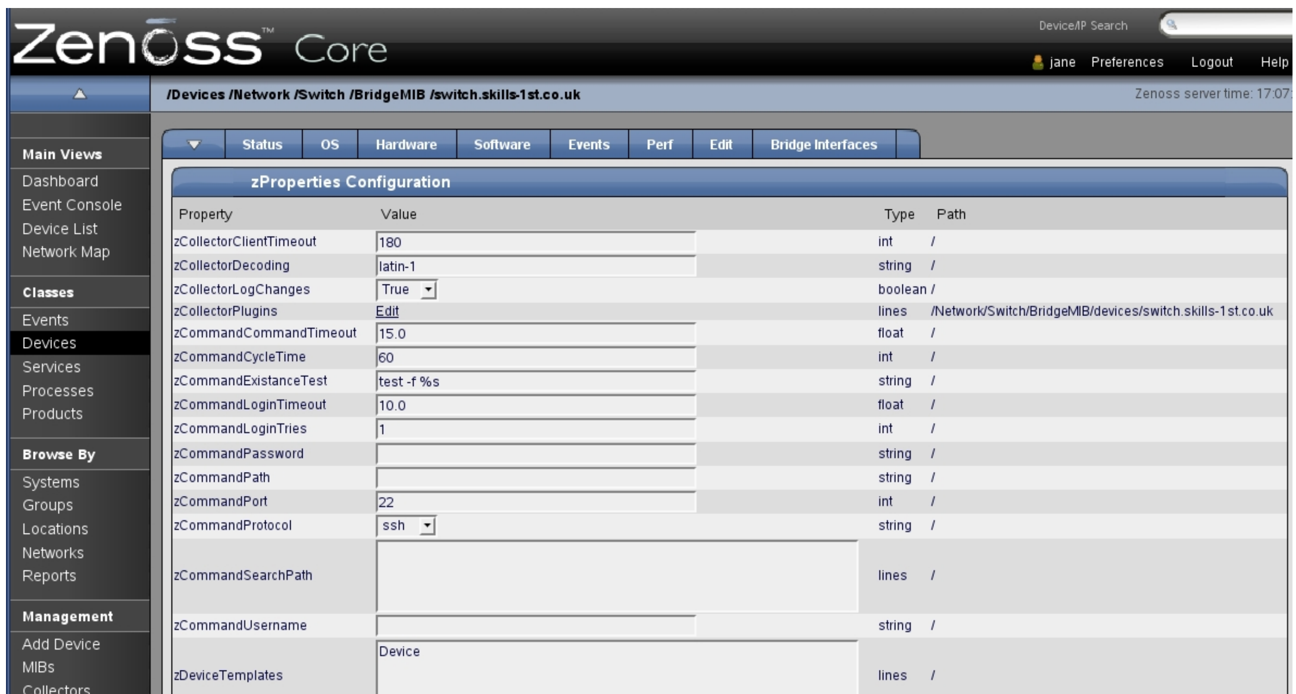

Figure 6: Modify the zCollectorPlugins zProperty to activate modeler plugins

As is usual with Zenoss, modeler plugins should be assigned as high as possible in the device class hierarchy to prevent unnecessary configuration and all sub device classes and devices will inherit that property; modeler plugins can always be deconfigured for a specific device if necessary.

Earlier versions of Zenoss seemed to need the dropdown *Manage - > Push Changes* option to be run for new configuration to take effect but with Zenoss 2.4 this appears to be unnecessary.

To run the modeler plugin on demand for a specific device, use the dropdown menu and *Manage -> Model Device*. Alternatively, for a specific device called switch.skills-1st.co.uk, use the following command line. The *-v 10* turns on debugging to loglevel 10 (the highest level).

```
zenmodeler run -v 10 -d switch.skills-1st.co.uk
```

You should see each of the modeler plugins listed and some results from each plugin.

Performance data to be collected is specified using Zenoss **templates**. As with modeler plugins, templates can be assigned either to a device class hierarchy or to a specific device but the definition of these templates, the RRD databases that contain the data and the daemons that collect the data are entirely separate from the configuration data collection mechanism. If you can access performance data using either SNMP or ssh then, typically, there is no need to write new code to collect performance data.

Modeler plugins are run by the zenmodeler daemon whereas SNMP performance template data is collected by zenperfsnmp and ssh-driven performance data is collected by the zencommand daemon. Another significant difference between modeler plugins and performance templates is that the configuration data collected by a modeler plugin will not trigger any “Component Type” status changes on a device's

main *Status* page, neither will any events be generated; however, if thresholds set in performance templates are exceeded or if performance data collection fails, then indications will show on the *Status* page and events can be seen and customised, as shown in Figure 7.

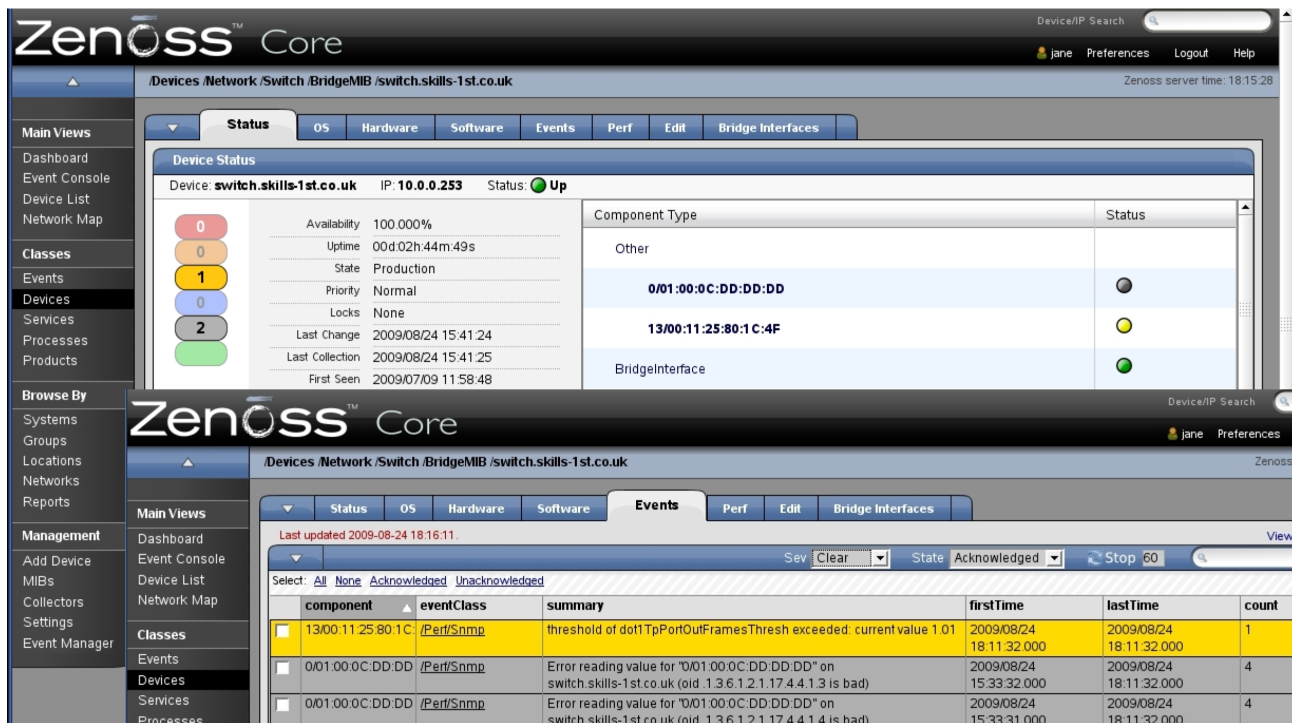

Figure 7: Status page for switch with thresholds set in performance template

4.1.2 The Zope Object Database (ZODB)

Zenoss is developed in Python using the open source Zope web application server – see http://www.zope.org/WhatIsZope for more information.

The Zope Object Database (ZODB) is an object-oriented Configuration Management Database (CMDB) used by Zope to store Python objects and their states; modeler plugins maintain information about devices and their configuration in the ZODB.

Zenoss uses ZEO, which is a layer between Zope and the ZODB. ZEO allows for multiple Zope servers to connect to the same ZODB. The ZODB is started and stopped by **zeoctl** . Note that the Zenoss documentation tends to use ZODB and ZEO interchangeably.

One way to get a feel for what is in the ZODB database and what Zope provides, is to point your browser at:

```
http://<zenoss server>:8080/zport/dmd/manage
```

You will need to authenticate yourself as a Zenoss user with Manager privileges if you have not already done so. The resulting screens allow you to explore the Zenoss **objects** (such as devices, event classes and MIBs) and also to display the **instances** of those objects (such as switch.skills-1st.class.example.org and BRIDGE-MIB).

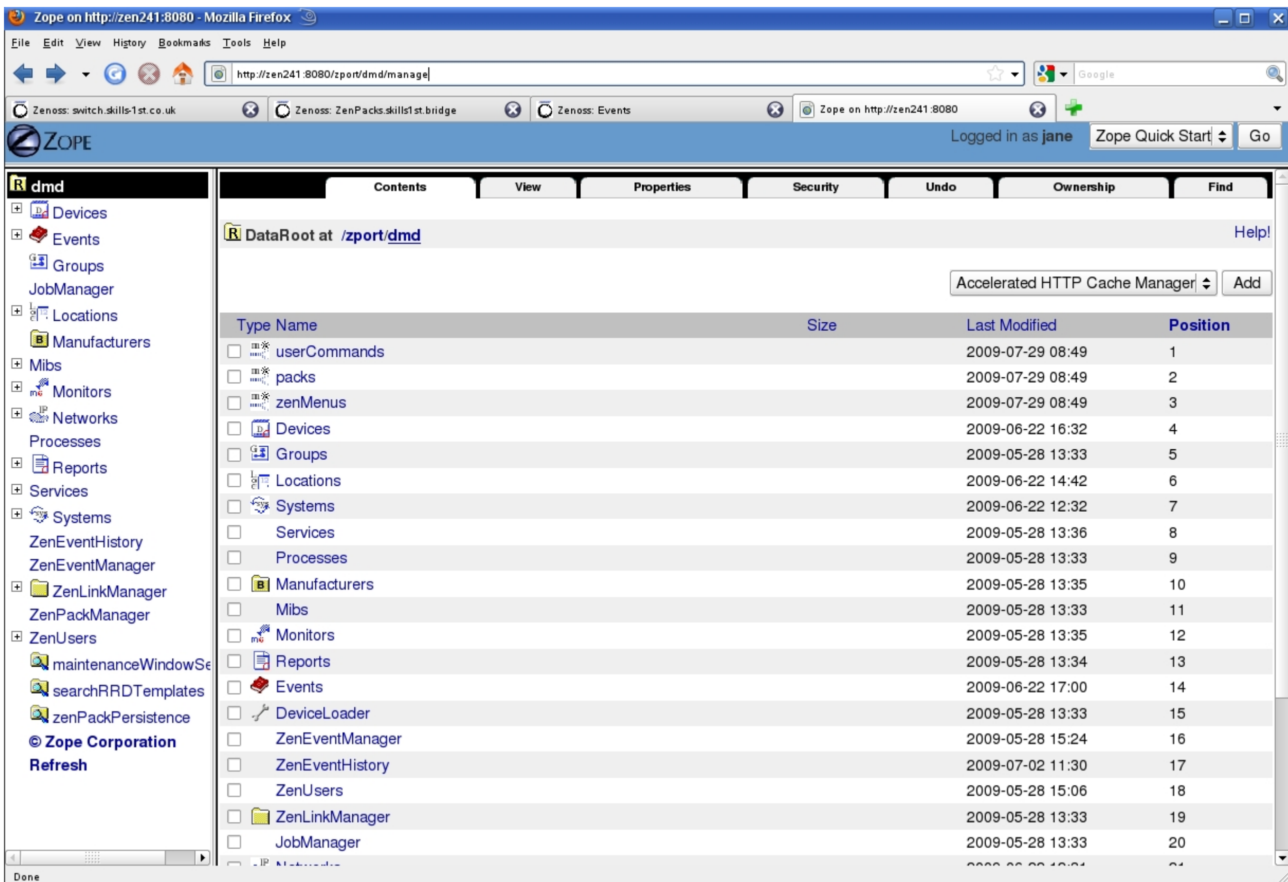

Figure 8: Accessing the ZODB database using the Zope interface

The top level of the ZODB database is *zport/dmd* (where dmd stands for Device Management Database). Note that the Zenoss Developer's Guide 2.4 has a very helpful glossary at the back which explains many of Zope's terms. If you omit the *manage* from the URL shown above, you will simply get to the standard Zenoss dashboard; adding *manage* provides access to the underlying Zope.

4.1.3 Coding techniques and terminology

When developing ZenPack code (in fact when administering Zenoss in any way), always ensure you are logged on as the *zenoss* user. When Zenoss is installed, this user is created but will be setup such that you cannot login directly as zenoss; you need to su to root and then use:

```
su - zenoss
```

to switch to the zenoss user.

If Python code is to be written, be aware that Python is very white-space sensitive. Program constructs such as if-then-else, while loops, for loops and many other coding elements depend on white space indentation (and the same number of spaces for the same level of the construct). If testing Python with the Zenoss-provided zendmd utility, the same white-space rules must be obeyed.

If a ZenPack is going to support new types of devices then a new **Python object class** needs to be created to describe the unique features of this device type. As with all object-oriented code, the new class can (and probably should) inherit some characteristics from its parent object class in a class hierarchy. Thus, the ZenPack discussed in this paper will create a new device class called *BridgeMIB*, which inherits from the standard device class */Devices/Network/Switch*; the unique characteristics of such a device are coded in a Python file in the base directory of the ZenPack (*/usr/local/zenoss/zenoss/local/jane/ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/BridgeDevice.py*).

A new device class is associated with a Python class through the *zPythonClass* zProperty (note that you do **not** specify the zPythonClass as a normal filesystem path but as a dotted class path from the ZenPack ie. *ZenPacks.skills1st.bridge.BridgeDevice* represents the file *BridgeDevice.py* (but don't include the *.py*) under the Zenoss ZenPacks directory *ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge* . More details on this later.

Standard object classes, such as *Device*, *OSComponent* and *IpInterface* can be found under *$ZENHOME/Products/ZenModel* . Note that each object class definition will have two files. The *.py* file is the Python source code; the *.pyc* file is the compiled Python code. There is no need to manually compile any Python code for Zenoss as this will be done automatically, as required.

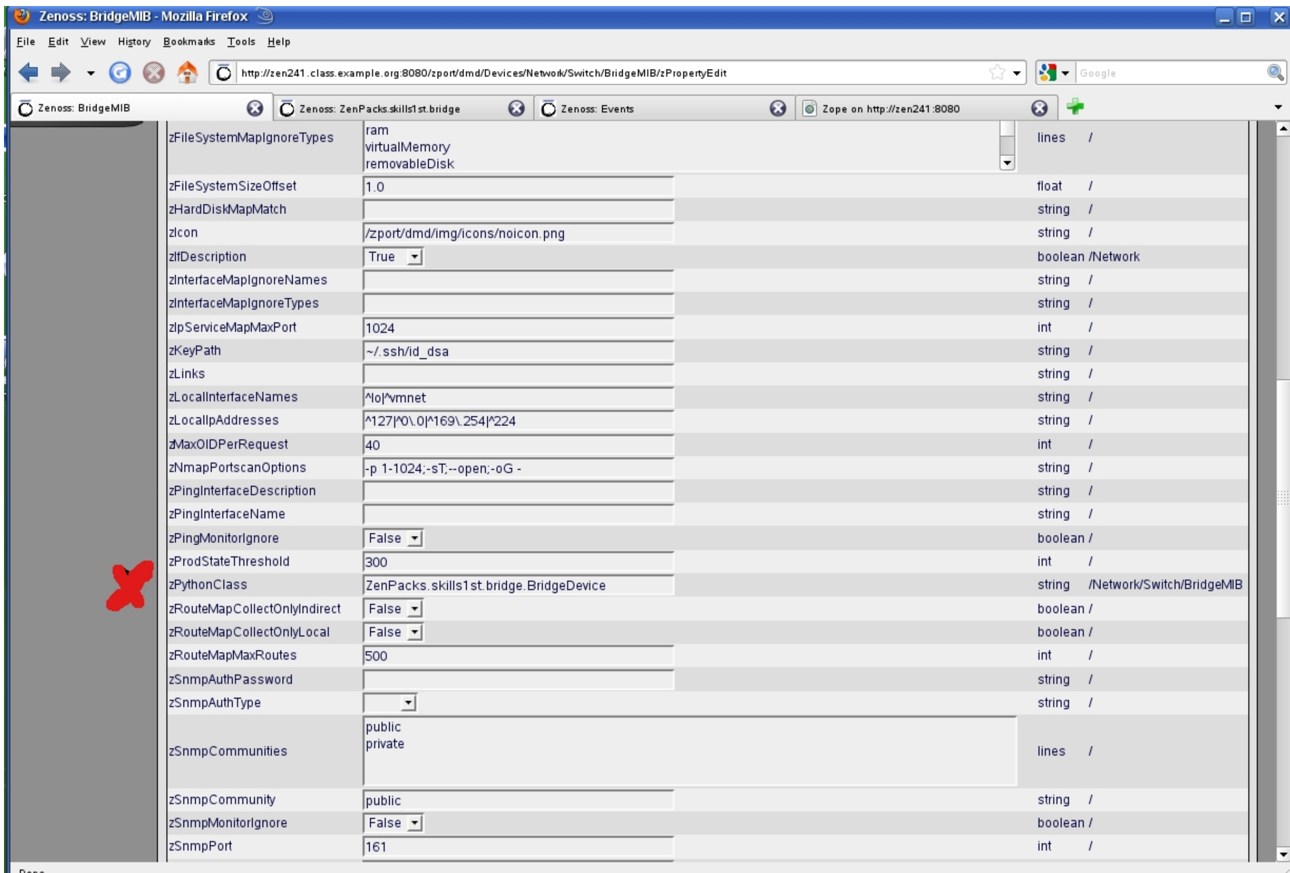

Figure 9: Associating a device class (BridgeMIB) with a Python class

Object classes that represent devices can have **relationships** with other classes. For example, *$ZENHOME/Products/ZenModel/Device.py,* which defines the base object class for devices, specifies a number of relationships as shown in Figure 10.

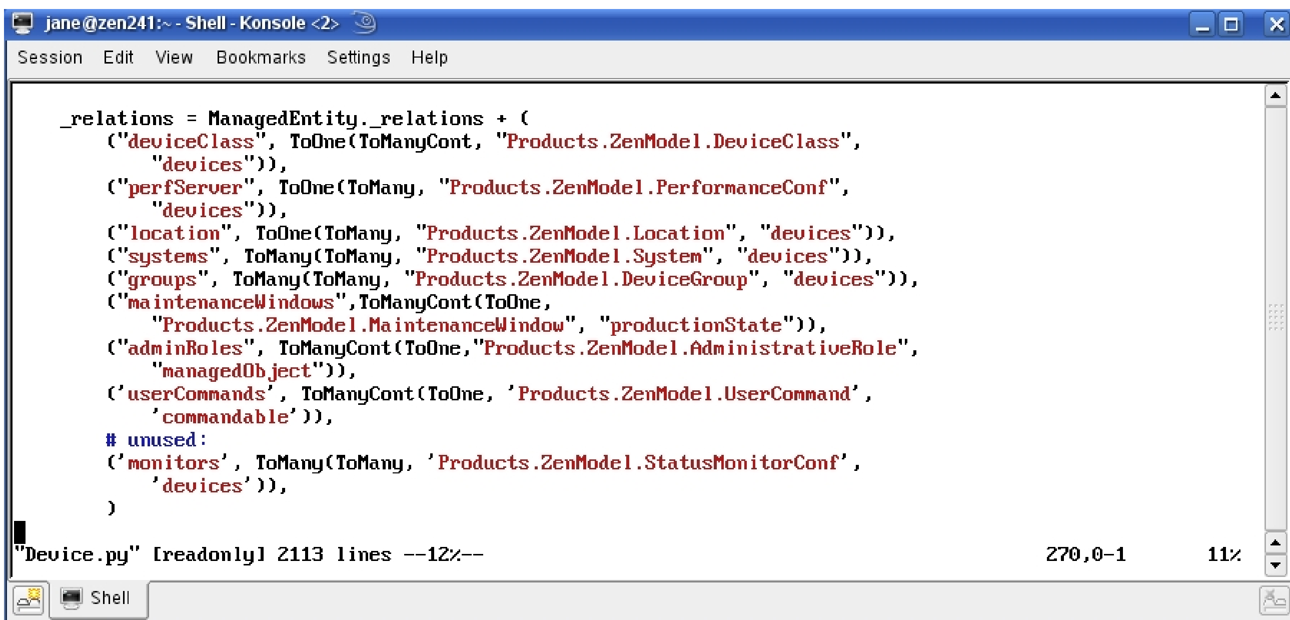

Figure 10: Relationships defined in Device.py

 © Skills 1st Ltd

Look at Chapter 9 of the Zenoss Developer's Guide 2.4 for details on the different types of relationships. Fundamentally, the code shown in Figure 10 is saying:

- A Device has a ToOne relationship with the object class DeviceClass (ie. any specific device can only belong to one device class)
- A Device has a ToOne relationship with the object class PerformanceConf (ie. any specific device will have only one performance data collector associated with it)
- A Device has a ToOne relationship with the object class Location (ie. any specific device can only be assigned to a single location)
- A Device has a ToMany relationship with the object class System (ie. any specific device can be assigned to several System groupings)
- A Device has a ToMany relationship with the object class DeviceGroup (ie. any specific device can be assigned to several Groups)
- A Device has a ToManyCont relationship with the object class MaintenanceWindow (ie. any specific device can contain several maintenance windows)
- A Device has a ToManyCont relationship with the object class AdministrativeRole (ie. any specific device can contain several administrative roles)
- A Device has a ToManyCont relationship with the object class UserCommand (ie. any specific device can contain several user commands)
- A Device has a ToMany relationship with the object class StatusMonitorConf (ie. any specific device can have several status monitors associated with it)

The syntax of the relationship statement seems rather perverse. Taking the first relationship from Device.py as an example:

```
("deviceClass", ToOne(ToManyCont, "Products.ZenModel.DeviceClass", "devices"))
```

- All relationships in this file are for the object class being defined, ie. Device in $ZENHOME/Products/ZenModel/Device.py
- The first field, *deviceClass* is the name of **this** relationship
- The relationship is a ToOne between Device and DeviceClass
- There is a corresponding relationship between DeviceClass and Device
    - The file $ZENHOME/Products/ZenModel/DeviceClass.py must contain this corresponding relationship (see Figure 11)
    - The relationship is a ToManyCont ie. a DeviceClass can contain many devices

- The name of the relationship defined in $ZENHOME/Products/ZenModel/DeviceClass.py is the last field, ie. *devices*

```
class DeviceClass(DeviceOrganizer, ZenPackable, TemplateContainer):
    """
    DeviceClass is a device organizer that manages the primary classification
    of device objects within the Zenoss system.  It manages properties
    that are inherited through acquisition that modify the behavior of
    many different sub systems within Zenoss.
    It also handles the creation of new devices in the system.
    """

    # Organizer configuration
    dmdRootName = "Devices"

    manageDeviceSearch = DTMLFile('dtml/manageDeviceSearch',globals())
    manageDeviceSearchResults = DTMLFile('dtml/manageDeviceSearchResults',
                                        globals())

    portal_type = meta_type = event_key = "DeviceClass"

    default_catalog = 'deviceSearch'

    _properties = DeviceOrganizer._properties + (
                    {'id':'devtypes', 'type':'lines', 'mode':'w'},
                   )

    _relations = DeviceOrganizer._relations + ZenPackable._relations + \
                TemplateContainer._relations + (
        ("devices", ToManyCont(ToOne,"Products.ZenModel.Device","deviceClass")),
        )
"DeviceClass.py" [readonly] 928 lines --9%--                      90,9           6%
```

Figure 11: $ZENHOME/Products/ZenModel/DeviceClass.py showing corresponding relationship with Device

Note that there is a specific relationship type when an object **contains** another object. Better examples exist in $ZENHOME/Products/ZenModel/OperatingSystem.py where an Operating System may contain many interfaces, routes, ipservices, winservices, processes, filesystems and software packages.

```
    _relations = Software._relations + (
        ("interfaces", ToManyCont(ToOne,
            "Products.ZenModel.IpInterface", "os")),
        ("routes", ToManyCont(ToOne, "Products.ZenModel.IpRouteEntry", "os")),
        ("ipservices", ToManyCont(ToOne, "Products.ZenModel.IpService", "os")),
        ("winservices", ToManyCont(ToOne,
            "Products.ZenModel.WinService", "os")),
        ("processes", ToManyCont(ToOne, "Products.ZenModel.OSProcess", "os")),
        ("filesystems", ToManyCont(ToOne,
            "Products.ZenModel.FileSystem", "os")),
        ("software", ToManyCont(ToOne, "Products.ZenModel.Software", "os")),
    )

"OperatingSystem.py" [readonly] 636 lines --9%--                  62,0-1         7%
```

Figure 12: ToManyCont relationships for the OperatingSystem object class

Where a container relationship exists, this often leads to a requirement to be able to conveniently display data about those contained components. For example, when viewing most device types through the GUI, there is an *OS* tab which shows data for these contained objects.

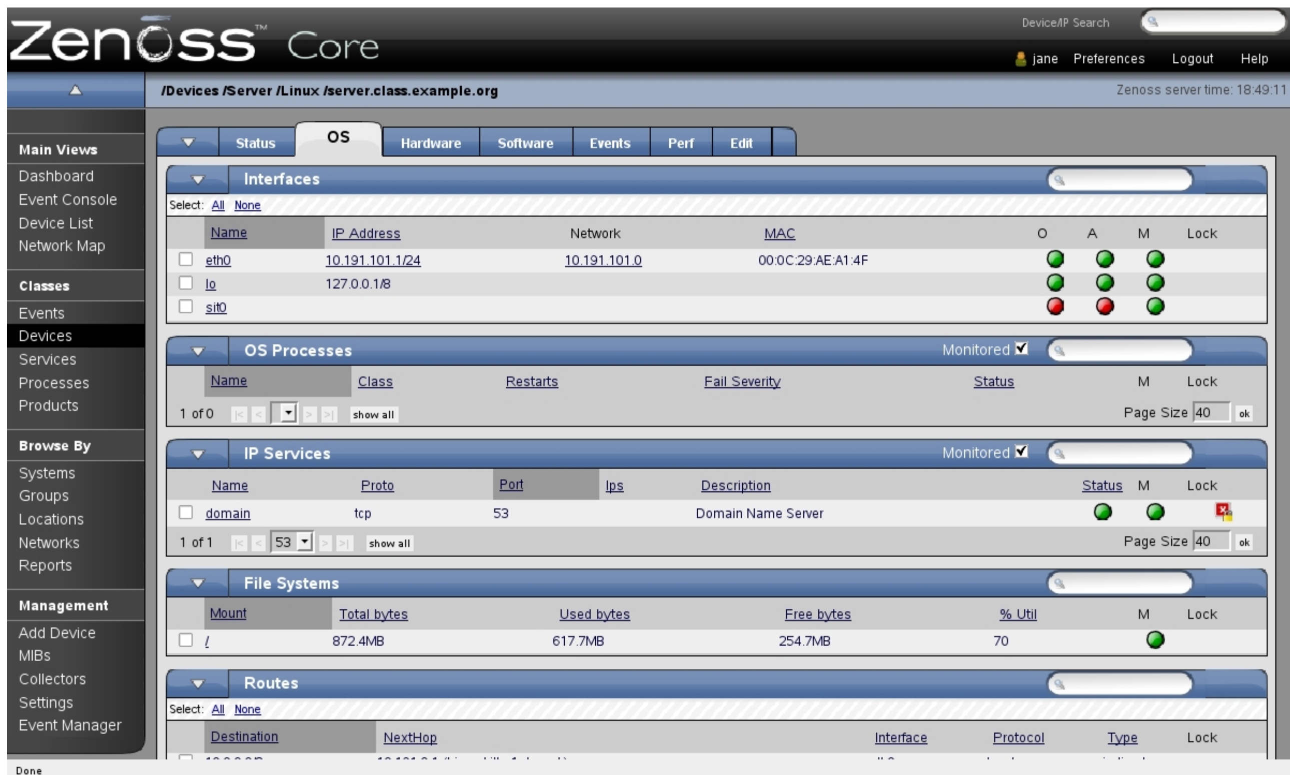

Figure 13: The OS tab for a device showing data for contained relationships such as interfaces, processes, filesystems, etc

Web pages for displaying data are defined in **skins** files. The skins files for the standard Zenoss objects are in the *$ZENHOME/Products/ZenModel/skins/zenmodel* directory and all have a *.pt* file extension (for Page Template). See the Zenoss Developer's Guide 2.4, Chapter 13 for details on writing skins files. You can use a mixture of:

- HyperText Markup Language (HTML)
- Cascading Style Sheets (CSS)
- Zope 2, Zope Page Templates (ZPT) and the Template Attribute Language (TAL)
- ZPT and Macro Expansion for TAL (METAL)
- JavaScript / **A**synchronous **J**avaScript **A**nd **X**ML (AJAX))
- Yahoo User Interface (YUI) Library and Mochikit

The file that defines the page for the OS tab, shown above in Figure 13, is *$ZENHOME/Products/ZenModel/skins/zenmodel/deviceOsDetail.pt*. It defines a form containing a table for each type of component, where the data to populate the table comes from the ZODB database.

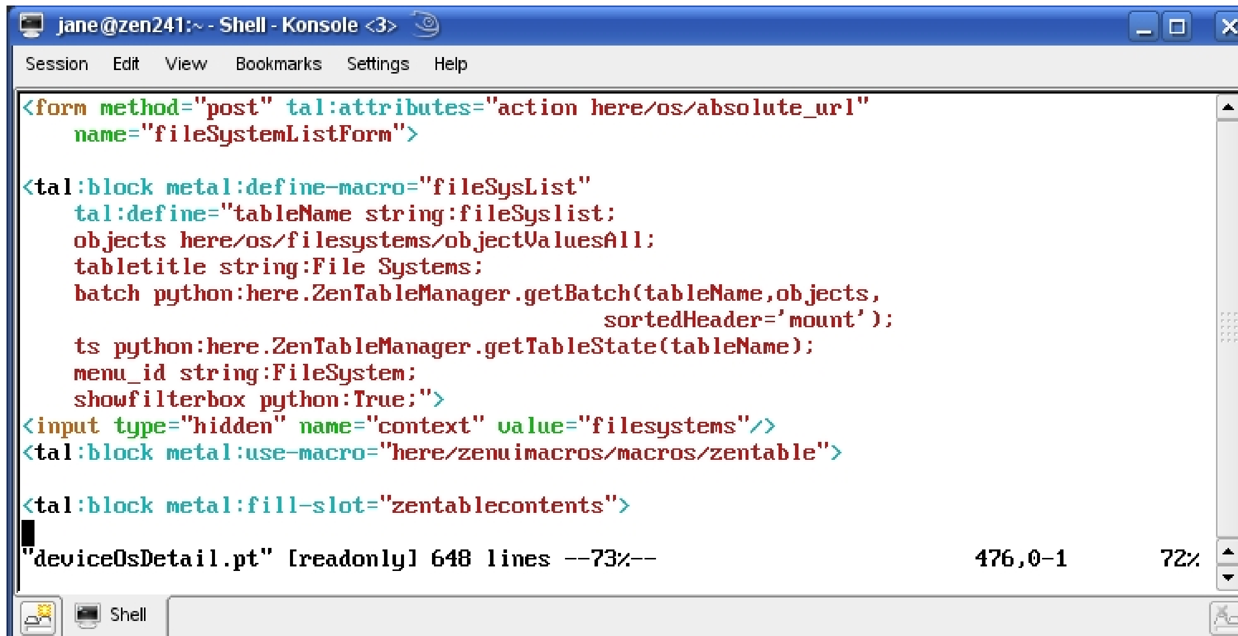

Figure 14: deviceOsDetail.pt skin file with definition of form for displaying filesystem information

The key line to note in Figure 14 is:

```
objects here/os/filesystem/objectValuesAll;
```

where *here* is the device in question (such as server.class.example.org), *os* is the Operating System object on the device, which in turn contains the *filesystem* object. *objectValuesAll* will return a table of data with one row for each filesystem on the device.

The layout of the table, including header columns and data columns can be very finely controlled. The first half of Figure 15 defines the table header columns; the middle 5 lines shows a check to ensure that data does actually exist to display; the next 3 lines (with *odd* and *even* in them) ensures that the rows of the table will have alternating light and dark backgrounds; and the rest of the screenshot is the start of the data values to populate the filesystem table. The intricacies of skins files will be examined in more detail later.

jane@zen241:~ - Shell - Konsole <3>

Session Edit View Bookmarks Settings Help

```
<!-- BEGIN TABLE CONTENTS -->
    <tr tal:condition="objects">
        <th class="tableheader" width="20"></th>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'mount','Mount')">Mount
        </th>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'totalBytes','Total bytes')">Total Bytes
        </th>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'usedBytes','Used bytes')">Used Bytes
        </th>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'freeBytes','Free bytes')">Free Bytes
        </th>
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'capacity','% Util')">% Util
        </th>
        <!--
        <th tal:replace="structure python:here.ZenTableManager.getTableHeader(
                    tableName,'storageDevice','Storage Device')">Device
        </th>
        -->
        <th class="tableheader" align="center" width="30">M</th>
        <th class="tableheader" align="center" width="60">Lock</th>
    </tr>
    <tr tal:condition="not:objects">
        <th class="tableheader" align="left">
            No File Systems
        </th>
    </tr>
    <tal:block tal:repeat="fsys batch">
    <tr tal:define="odd repeat/fsys/odd"
        tal:attributes="class python:test(odd, 'odd', 'even')">

      <td class="tablevalues" align="center">
           <input type="checkbox" name="componentNames:list"
                  tal:attributes="value fsys/getRelationshipManagerId"/>
      </td>
      <td class="tablevalues">
          <tal:block
           tal:content="structure python:fsys.urlLink(text=fsys.mount,
            attrs={'class':'tablevalues'})"/>
      </td>
      <td class="tablevalues"
"deviceOsDetail.pt" [readonly] 648 lines --80%--                  521,7          78%
```

Shell

Figure 15: deviceOsDetail.pt showing layout of table for filesystems data

So what links the device object class with the skins file that displays web pages of data relating to a device? This is coded in the object class file, after the relationship statements. Each tab required for the object has a stanza defining its *id*, *name*, *action* and *permissions;* it is the *action* field that specifies the name of the skins file (without the *.pt*). Compare the (incomplete) definitions in Figure 16 with the tabs shown for a device in Figure 13. The *name* field gives the name shown on the tab; the *action* field should match with the name of a skins file (without the *.pt*) in *$ZENHOME/Products/ZenModel/skins/zenmodel.*

jane@zen241:~ - Shell - Konsole <2>

Session Edit View Bookmarks Settings Help

```
    # Screen action bindings (and tab definitions)
    factory_type_information = (
        {
            'id'             : 'Device',
            'meta_type'      : 'Device',
            'description'    : """Base class for all devices""",
            'icon'           : 'Device_icon.gif',
            'product'        : 'ZenModel',
            'factory'        : 'manage_addDevice',
            'immediate_view' : 'deviceStatus',
            'actions'        :
            (
                { 'id'            : 'status'
                , 'name'          : 'Status'
                , 'action'        : 'deviceStatus'
                , 'permissions'   : (ZEN_VIEW, )
                },
                { 'id'            : 'osdetail'
                , 'name'          : 'OS'
                , 'action'        : 'deviceOsDetail'
                , 'permissions'   : (ZEN_VIEW, )
                },
                { 'id'            : 'hwdetail'
                , 'name'          : 'Hardware'
                , 'action'        : 'deviceHardwareDetail'
                , 'permissions'   : (ZEN_VIEW, )
                },
                { 'id'            : 'swdetail'
                , 'name'          : 'Software'
                , 'action'        : 'deviceSoftwareDetail'
                , 'permissions'   : (ZEN_VIEW, )
                },
                { 'id'            : 'events'
                , 'name'          : 'Events'
                , 'action'        : 'viewEvents'
                , 'permissions'   : (ZEN_VIEW, )
                },
#                { 'id'            : 'historyEvents'
#                , 'name'          : 'History'
#                , 'action'        : 'viewHistoryEvents'
#                , 'permissions'   : (ZEN_VIEW, )
#                },
                { 'id'            : 'perfServer'
                , 'name'          : 'Perf'
                , 'action'        : 'viewDevicePerformance'
                , 'permissions'   : (ZEN_VIEW, )
                },
"Device.py" [readonly] 2113 lines --15%--                    317,17        13%
```

Figure 16: Device.py object class file showing the action filenames for each tab

4.1.4 Databases, Daemons and Directories

To summarise this “Basic Principles” section, here are a couple of diagrams showing the architecture of Zenoss.

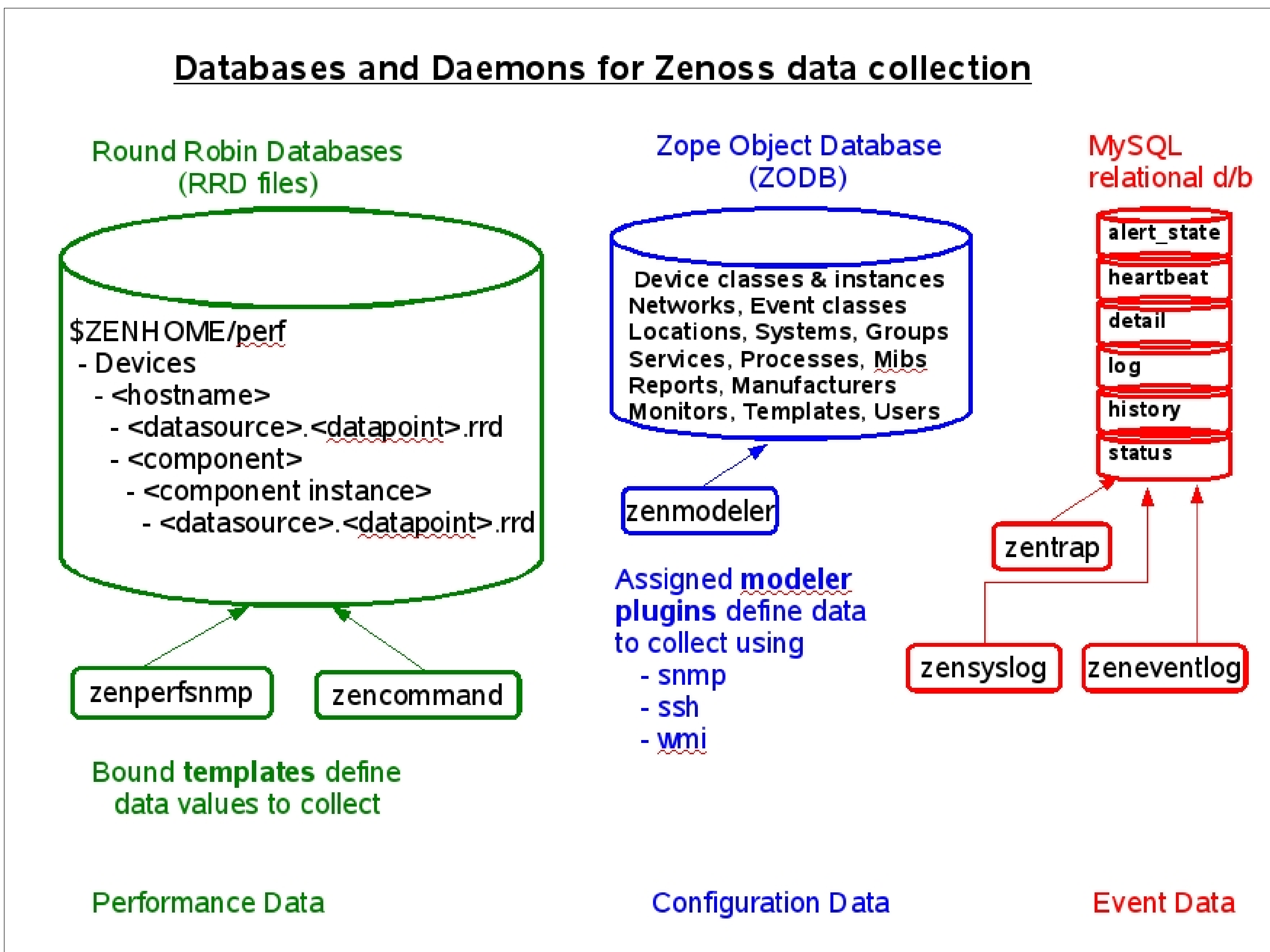

Figure 17: Databases and Daemons for Zenoss data collection

Figure 17 shows the 3 different databases used by Zenoss:

- Performance data is held in Round Robin Database (RRD) files under $ZENHOME/perf
- Configuration data is held in the Zope Object Database (ZODB)
- Event data is held in a MySQL database

Performance data is typically collected at frequent intervals (SNMP data is collected every 5 minutes, by default). Templates define **datasources** and **datapoints** to be collected, where a datasource includes the source type (such as SNMP) and the OID to collect (in the case of SNMP). For SNMP data, the datapoint will have the same name as the datasource. If data is collected using ssh then the datasource type will be *COMMAND* and the polling interval can also be specified. Since an ssh command may return several datapoints, each has to be specified with a unique name. SNMP performance data is collected by the **zenperfsnmp** daemon whilst ssh data is collected by **zencommand**.

Performance data is stored under $ZENHOME/perf/Devices with a separate directory for each device. Performance values for the device itself will be under this hostname subdirectory, with the format <datasource>.<datapoint>.rrd; for example:

```
$ZENHOME/perf/Devices/zen241.class.example.org/laLoadInt1_laLoadInt1.rrd
$ZENHOME/perf/Devices/bino.skills-1st.co.uk/procs_linuxNum.rrd
```

If the object class of the device has contained components, such as *os*, which itself contains *filesystems* objects and *interfaces* objects, then the directory hierarchy under the hostname is extended to reflect and store the component data. Thus, interface information for the interface called eth1 on the device bino.skills-1st.co.uk would be stored in *$ZENHOME/perf/Devices/ bino.skills-1st.co.uk/os/interfaces/eth1* and would include datafiles such as *ifInOctets_ifInOctets.rrd*.

Note that a template must actually be **bound** to a device or device class before data collection will be effected.

Configuration data is collected by the **zenmodeler** daemon, each device or device class having been configured for one or more **modeler plugins**. The standard modeler plugins include SNMP, WMI and ssh as data collection protocols. Configuration data is stored in the Zope Object Database (ZODB).

Event data is stored in a MySQL relational database (that is installed and configured automatically when Zenoss is installed). The database has 6 tables:

- status
- history
- log
- detail
- heartbeat
- alert_status

The active events are in the status table whereas closed events are in the history table.

Events are generated and inserted into the database by various of the Zenoss daemons (such as **zenping**, **zenstatus** and **zenperfsnmp**). External events can also be captured and inserted from syslogs by the **zensyslog** daemon, from SNMP TRAPs by the **zentrap** daemon, and from Windows event logs by the **zeneventlog** daemon.

Figure 18 shows the directory structure for performance and configuration data.

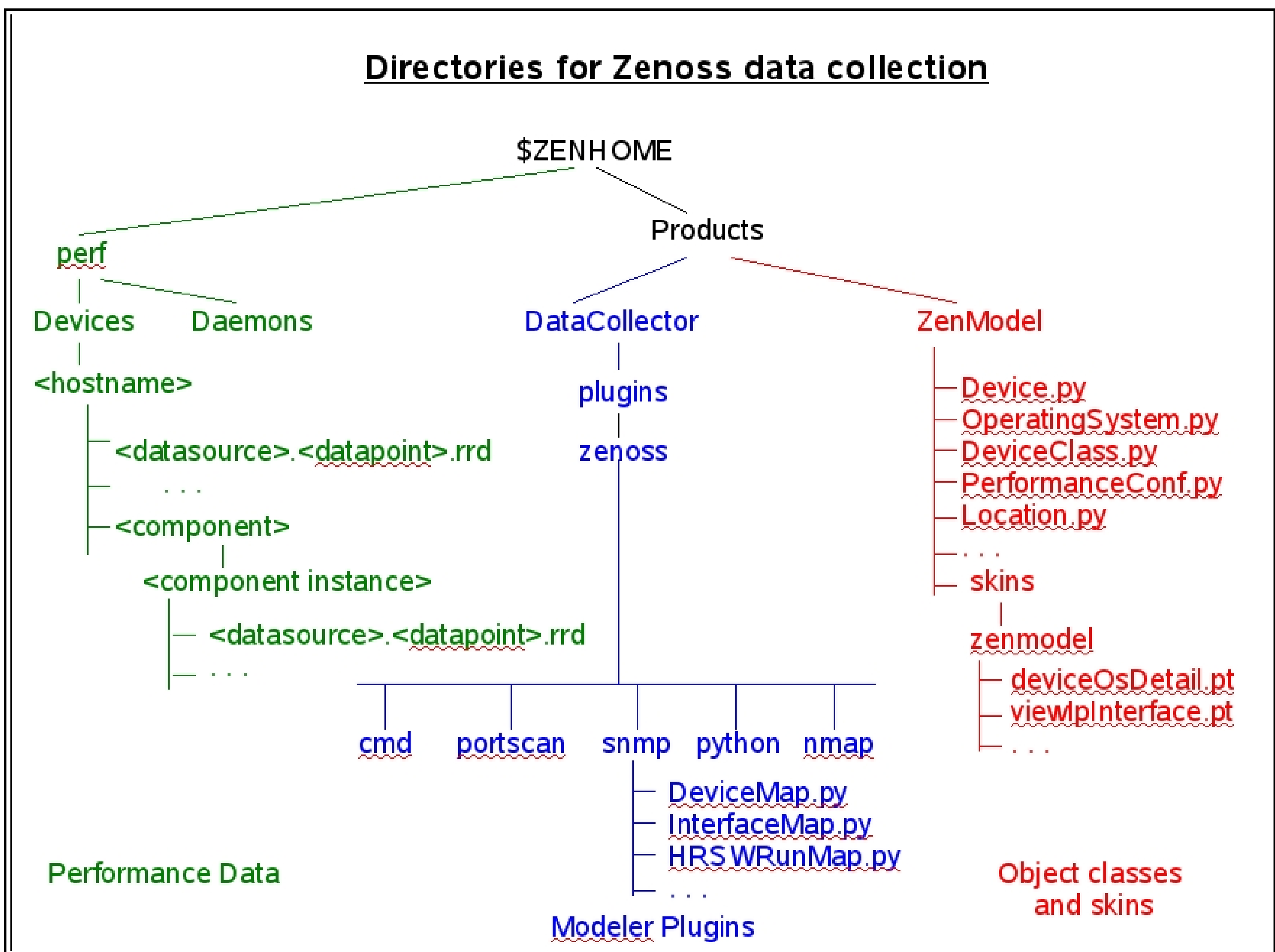

Figure 18: Directory hierarchy for Zenoss data collection

4.2 Requirements for the sample ZenPack

To illustrate the different elements of ZenPacks, a sample ZenPack will be created to get extra information from switch devices that support the BRIDGE MIB (as defined by RFCs 1493 and 4188). The BRIDGE MIB provides information for each port on a switch, including the MAC address(es) that have been seen connected at the other end of the switch port; thus it is possible to build some ideas of layer 2 connectivity.

The main information that the BRIDGE MIB will supply to the ZenPack comes from the Forwarding Database for Transparent Bridges table (OID .1.3.6.1.2.1.17.4.3.1).

```
Session   Edit   View   Bookmarks   Settings   Help

-- ---------------------------------------------------------- --
--  The Forwarding Database for Transparent Bridges
-- ---------------------------------------------------------- --

dot1dTpFdbTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF Dot1dTpFdbEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "A table that contains information about unicast
        entries for which the bridge has forwarding and/or
        filtering information.  This information is used
        by the transparent bridging function in
        determining how to propagate a received frame."
    ::= { dot1dTp 3 }

dot1dTpFdbEntry OBJECT-TYPE



    SYNTAX      Dot1dTpFdbEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "Information about a specific unicast MAC address
        for which the bridge has some forwarding and/or
        filtering information."
    INDEX   { dot1dTpFdbAddress }
    ::= { dot1dTpFdbTable 1 }

Dot1dTpFdbEntry ::=
    SEQUENCE {
        dot1dTpFdbAddress
            MacAddress,
        dot1dTpFdbPort
            Integer32,
        dot1dTpFdbStatus
            INTEGER
    }
```

Figure 19: BRIDGE MIB - Forwarding Database for Transparent Bridges section

The three leaf-node OIDs are:

- dot1dTpFdbAddress of type MacAddress
- dot1dTpFdbPort of type Integer32
- dot1dTpFdbStatus – an enumerated INTEGER type where :
    - other(1),
    - invalid(2),
    - learned(3),
    - self(4),
    - mgmt(5)

For this ZenPack sample, only ports that have a status of “learned” (3) are going to be considered as active (ie. traffic has actively been seen going out of that port to a MAC address).

To make matters more confusing, the value supplied by the BRIDGE MIB for dot1dTpFdbPort for some switches, does not match obvious port numbers. For example, a Cisco Catalyst 2900 has 24 physical ports (actually labelled 1 – 24). The BRIDGE MIB reports the first physical port as dot1dTpFdbPort = 13, the second as 14, and so on. To help a little with this confusion, the BRIDGE MIB provides the Generic Bridge Port Table (OID .1.3.6.1.2.1.17.1.4.1) whose first two leaf-node OIDs are:

- dot1dBasePort - "The port number of the port for which this entry contains bridge management information" – ie. the same port number as reported by dot1dTpFdbPort
- dot1dBasePortIfIndex - "The value of the instance of the ifIndex object, defined in IF-MIB, for the interface corresponding to this port." In other words, this value provides a cross-reference between BRIDE MIB port references and their interfaces reported by the standard MIB-2 interface table. For example, the port that is physically labelled 1, reports dot1dTpFdbPort=13, dot1dBasePortIfIndex=2 and information from the MIB-2 interface table for this port reports ifIndex=2 with the corresponding ifDescr=”FastEthernet0/1” - I do hope that's clear!

So, to build any semblance of a layer 2 topology, we need to coordinate several pieces of information from the BRIDGE MIB and from MIB-2. The target is to be able to display information as shown in Figure 20.

Zenoss Core

/Devices /Network /Switch /BridgeMIB /switch.skills-1st.co.uk — Zenoss server time: 13:21:20

Bridge Interfaces Table

Port Name	Port Number	Port Interface Index	Remote Address	Remote IP Address	Remote Hostname	Remote Interface Description	Port Status Value	Port Status
13/00:0C:41:9D:D3:81	13	2	00:0C:41:9D:D3:81	[]	[]	[]	Learned (3)	
13/00:0E:35:64:72:A7	13	2	00:0E:35:64:72:A7	[]	[]	[]	Learned (3)	
13/00:11:25:80:1C:4F	13	2	00:11:25:80:1C:4F	['10.0.0.121']	['bino.skills-1st.co.uk']	['eth1']	Learned (3)	
40/00:04:C1:9C:90:C0	40	-1	00:04:C1:9C:90:C0	['10.0.0.253']	['switch.skills-1st.co.uk']	['VLAN1']	Not active (4)	
40/00:04:C1:9C:90:C1	40	-1	00:04:C1:9C:90:C1	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_1']	Not active (4)	
40/00:04:C1:9C:90:C2	40	-1	00:04:C1:9C:90:C2	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_2']	Not active (4)	
40/00:04:C1:9C:90:C3	40	-1	00:04:C1:9C:90:C3	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_3']	Not active (4)	
40/00:04:C1:9C:90:C4	40	-1	00:04:C1:9C:90:C4	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_4']	Not active (4)	
40/00:04:C1:9C:90:C5	40	-1	00:04:C1:9C:90:C5	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_5']	Not active (4)	
40/00:04:C1:9C:90:C6	40	-1	00:04:C1:9C:90:C6	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_6']	Not active (4)	
40/00:04:C1:9C:90:C7	40	-1	00:04:C1:9C:90:C7	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_7']	Not active (4)	
40/00:04:C1:9C:90:C8	40	-1	00:04:C1:9C:90:C8	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_8']	Not active (4)	
40/00:04:C1:9C:90:C9	40	-1	00:04:C1:9C:90:C9	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_9']	Not active (4)	
40/00:04:C1:9C:90:CA	40	-1	00:04:C1:9C:90:CA	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_10']	Not active (4)	
40/00:04:C1:9C:90:CB	40	-1	00:04:C1:9C:90:CB	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_11']	Not active (4)	

Figure 20: The Bridge Interfaces Table for a Catalyst 2900

This screenshot shows three active MAC addresses, two of which have been detected on the same port 13 (ifIndex 2, physically labelled port 1); this is perfectly likely as the port is connected to a router. Of these two MAC addresses, Zenoss is unable to provide remote information (other than the MAC itself) for the first address – this is because the remote device has not been discovered by Zenoss. The second MAC address seen down port 13 is for a device that Zenoss has discovered so the remote IP address, hostname and interface description has been supplied out of the ZODB database. Port number 22 (ifIndex 10, physically labelled port 9) is also active and connected to remote IP address 10.0.0.20, hostname taplow-20.skills-1st.co.uk, whose remote interface description is eth1. The rest of the ports are actually not connected so show a Port Status that is **not** 3.

In addition to getting port information from the BRIDGE MIB, it can also deliver its base bridge address as OID .1.3.6.1.2.1.17.1.1.0 and the total number of ports on the switch as OID .1.3.6.1.2.1.17.1.2.0. These values will be collected and displayed on the Status tab of a switch.

As shown in Figure 20, port information will be displayed in a new tab that will automatically be created for devices that support the BRIDGE MIB.

In addition to showing a table of ports, clicking on any port row will display performance information for that port as shown in Figure 21.

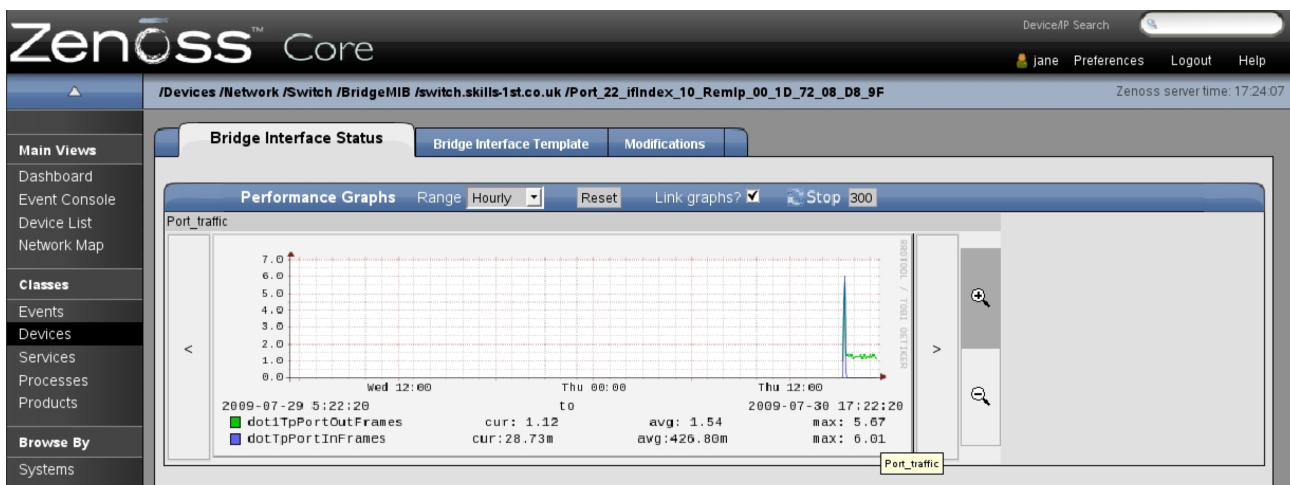

Figure 21: Performance graph for a selected switch port

The template that delivers switch port performance information can have whatever datapoints you wish to configure but the template **name** must match the object class of the device component (BridgeInterface in this case) – more of this later.

 © Skills 1st Ltd

4.3 Creating the sample ZenPack

It is essential to plan out the pieces of code required for a ZenPack and clearly document the names that will be used as many elements are referenced in other elements. Note that all names are case-sensitive.

4.3.1 Elements required and their names

This ZenPack is for devices that support the bridge MIB and it is created by Skills 1st, so the name of the ZenPack will be:

- **ZenPacks.skills1st.bridge**

This means that a directory hierarchy will automatically be created under ZenPacks.skills1st.bridge:

  - **ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge**

This directory will be referred to as the base directory of the ZenPack throughout this section, as it contains the object class files, the modeler directory and the skins directory.

A new device class will be used for such devices which is a subclass of the standard Switch device class. The device class is created through the GUI, simply by navigating to *Devices -> Network -> Switch* and using the Sub-Devices table drop-down menu to *Add New Organizer*. The new device class will be:

- **BridgeMIB**

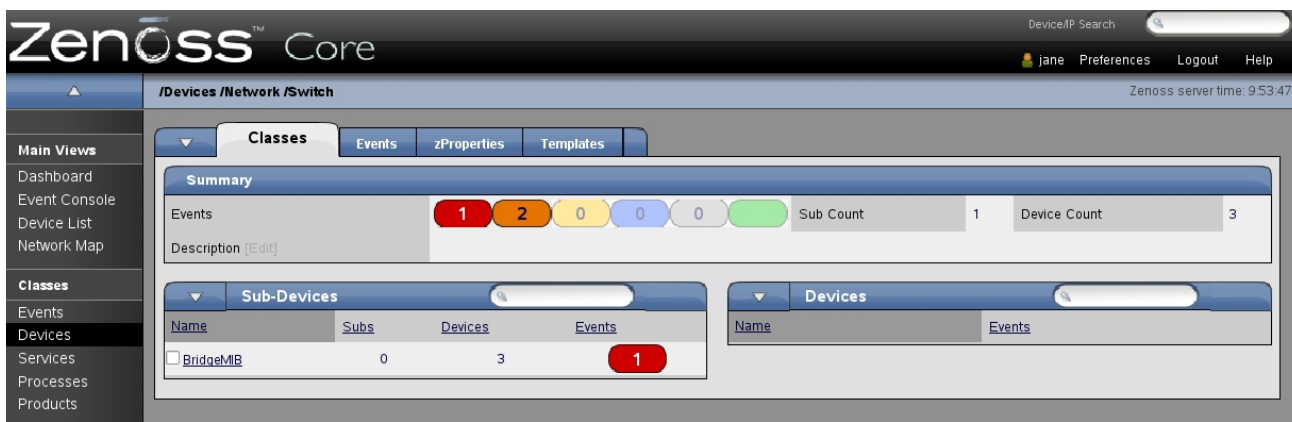

Figure 22: Creating a device class for BridgeMIB as a subclass of /Devices/Network/Switch

Note that this is a **device** class. It is not the **object** class file that specifies what makes such a device unique (relating the device class with the object class file will be discussed later).

The solution actually needs two new object class files; one for the device itself (**BridgeDevice**) and one to represent an interface on a bridge device (**BridgeInterface**). These files contain Python code and must exist in the base

directory of the ZenPack. The name of the file should reflect the name of the object class that is being defined; thus:

- **BridgeDevice.py** contains the line **class BridgeDevice(Device):**
- **BridgeInterface.py** contains the line

class BridgeInterface(DeviceComponent, ManagedEntity):

Within these object class files, the relationships between BridgeDevice and BridgeInterface will be specified (refer back to section 4.1.3 for information on relationships). Each relationship also has a name, distinct from the object class name so:

- A BridgeDevice object will have a relationship called **BridgeInt** defining a ToManyCont relationship with a BridgeInterface object (ie a BridgeDevice may contain many BridgeInterfaces).
- A BridgeInterface object will have a relationship called **BridgeDev** defining a ToOne relationship with a BridgeDevice object (ie. a BridgeInterface is associated with only one BridgeDevice).

Note that, by convention, the relationship name tends to reflect what is being related **to**.

Also note that some ZenPacks (especially older ones) define relationships in the __init__.py file of the base directory of the ZenPack. This procedure is also alluded to in the Zenoss Developer's Guide 2.4. My understanding, with recent versions of code, is that there is no requirement to modify any of the automatically-created __init__.py files if relationships are specified in object class files, as shown here.

Having created new object class files, modeler plugin code is required to populate the fields of these objects so the *modeler/plugins* directory under the ZenPack base directory contains:

- **BridgeDeviceMib.py**
- **BridgeInterfaceMib.py**

These files have Python code that use the standard SnmpPlugin collector to gather relevant SNMP data for the new objects. As discussed in section 4.1.1, modeler plugins are assigned to devices or device classes using the GUI with the *More -> Collector Plugins* drop-down menu.

The final elements required are the web pages to show information about the new objects – these are held in the *skins/ZenPacks.skills1st.bridge* directory under the ZenPack base directory and have a *.pt* extension:

- **BridgeDeviceDetail.pt**
- **viewBridgeInterface.pt**

4.3.2 SNMP data required

Fundamentally, a protocol is necessary to gather both configuration and performance data. This ZenPack uses SNMP for both. It is always advisable to check that devices do respond to SNMP using a basic (non-Zenoss) SNMP command utility. The format of the command depends on the version of SNMP for which the device is configured. Here are examples for SNMP versions 1, 2 and 3, using the net-snmp utility, to walk the SNMP MIB tree from the BRIDGE MIB Forwarding Table (.1.3.6.1.2.1.17.4.3.1) for a device called switch (the hostname can be anything you can ping so potentially short hostnames will work just as well as fully-qualified Domain Names). SNMP versions 1 and 2c have a community name of *fraclmye* configured for use with the Zenoss server. The SNMP V3 version uses *MD5* authentication, passphrase *fraclmyea*, and user *jane2*.

- `snmpwalk -v 1 -c fraclmye switch .1.3.6.1.2.1.17.4.3.1`
- `snmpwalk -v 2c -c fraclmye switch .1.3.6.1.2.1.17.4.3.1`
- `snmpwalk -v 3 - a MD5 -A fraclmyea -l authNoPriv -u jane2 switch .1.3.6.1.2.1.17.4.3.1`

Once basic SNMP communication is established, make sure that Zenoss device classes and/or devices have the correct SNMP parameters configured in their zProperties page.

The ZenPack will need two sets of table data from the BRIDGE MIB and two scalar values:

.1.3.6.1.2.1.7.4.3.1 (dot1dTpFdbEntry)	.1 (RemoteAddress) .2 (Port) .3 (PortStatus)
.1.3.6.1.2.1.7.1.4.1 (dot1dBasePortEntry)	.1 (BasePort) .2 (BasePortifIndex)

Table 4.1.: Table entries from the BRIDGE MIB for each port of a switch

.1.3.6.1.2.1.17.1.1.0	dot1dBaseBridgeAddress
.1.3.6.1.2.1.17.1.2.0	dot1dBaseNumPorts

Table 4.2.: Scalar entries from the BRIDGE MIB

Note that the *.0* is required on the end for the scalar MIB values.

Some of the values returned are shown in the following screenshots:

```
jane@zen241:~ - Shell - Konsole <3>
Session  Edit  View  Bookmarks  Settings  Help
s/ZenPacks.skills1st.bridge> snmpwalk -v 1 -c public  switch.skills-1st.co.uk .1.3.6.1.2.1.17.4.3.1.1
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.192 = Hex-STRING: 00 04 C1 9C 90 C0
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.193 = Hex-STRING: 00 04 C1 9C 90 C1
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.194 = Hex-STRING: 00 04 C1 9C 90 C2
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.195 = Hex-STRING: 00 04 C1 9C 90 C3
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.196 = Hex-STRING: 00 04 C1 9C 90 C4
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.197 = Hex-STRING: 00 04 C1 9C 90 C5
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.198 = Hex-STRING: 00 04 C1 9C 90 C6
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.199 = Hex-STRING: 00 04 C1 9C 90 C7
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.200 = Hex-STRING: 00 04 C1 9C 90 C8
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.201 = Hex-STRING: 00 04 C1 9C 90 C9
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.202 = Hex-STRING: 00 04 C1 9C 90 CA
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.203 = Hex-STRING: 00 04 C1 9C 90 CB
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.204 = Hex-STRING: 00 04 C1 9C 90 CC
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.205 = Hex-STRING: 00 04 C1 9C 90 CD
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.206 = Hex-STRING: 00 04 C1 9C 90 CE
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.207 = Hex-STRING: 00 04 C1 9C 90 CF
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.208 = Hex-STRING: 00 04 C1 9C 90 D0
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.209 = Hex-STRING: 00 04 C1 9C 90 D1
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.210 = Hex-STRING: 00 04 C1 9C 90 D2
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.211 = Hex-STRING: 00 04 C1 9C 90 D3
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.212 = Hex-STRING: 00 04 C1 9C 90 D4
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.213 = Hex-STRING: 00 04 C1 9C 90 D5
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.214 = Hex-STRING: 00 04 C1 9C 90 D6
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.215 = Hex-STRING: 00 04 C1 9C 90 D7
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.216 = Hex-STRING: 00 04 C1 9C 90 D8
SNMPv2-SMI::mib-2.17.4.3.1.1.0.12.41.149.80.111 = Hex-STRING: 00 0C 29 95 50 6F
SNMPv2-SMI::mib-2.17.4.3.1.1.0.12.65.157.211.129 = Hex-STRING: 00 0C 41 9D D3 81
SNMPv2-SMI::mib-2.17.4.3.1.1.0.14.53.100.114.167 = Hex-STRING: 00 0E 35 64 72 A7
SNMPv2-SMI::mib-2.17.4.3.1.1.0.17.37.128.28.79 = Hex-STRING: 00 11 25 80 1C 4F
SNMPv2-SMI::mib-2.17.4.3.1.1.1.0.12.0.0.0 = Hex-STRING: 01 00 0C 00 00 00
SNMPv2-SMI::mib-2.17.4.3.1.1.1.0.12.204.204.204 = Hex-STRING: 01 00 0C CC CC CC
SNMPv2-SMI::mib-2.17.4.3.1.1.1.0.12.204.204.205 = Hex-STRING: 01 00 0C CC CC CD
SNMPv2-SMI::mib-2.17.4.3.1.1.1.0.12.221.221.221 = Hex-STRING: 01 00 0C DD DD DD
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.0 = Hex-STRING: 01 80 C2 00 00 00
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.1 = Hex-STRING: 01 80 C2 00 00 01
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.2 = Hex-STRING: 01 80 C2 00 00 02
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.3 = Hex-STRING: 01 80 C2 00 00 03
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.4 = Hex-STRING: 01 80 C2 00 00 04
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.5 = Hex-STRING: 01 80 C2 00 00 05
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.6 = Hex-STRING: 01 80 C2 00 00 06
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.7 = Hex-STRING: 01 80 C2 00 00 07
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.8 = Hex-STRING: 01 80 C2 00 00 08
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.9 = Hex-STRING: 01 80 C2 00 00 09
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.10 = Hex-STRING: 01 80 C2 00 00 0A
SNMPv2-SMI::mib-2.17.4.3.1.1.1.128.194.0.0.11 = Hex-STRING: 01 80 C2 00 00 0B
```

Figure 23: Results from performing snmpwalk for the RemoteAddress values of the Port Forwarding table of the BRIDGE MIB

Note that the OID index (the numbers after mib-2.17.4.3.1.1 represent the MAC address in decimal; thus in the first response of:

```
SNMPv2-SMI::mib-2.17.4.3.1.1.0.4.193.156.144.192 = Hex-STRING: 00 04 C1 9C 90 C0
```

the RemoteAddress MAC is *00 04 C1 9C 90 C0* and the index is *0.4.193.156.144.192* where:

MAC Address	Index	
00	0	
04	4	
C1	12 x 16 + 1 = 193	
9C	9 x 16 + 12 = 156	and so on

```
jane@zen241:~ - Shell - Konsole <3>
Session  Edit  View  Bookmarks  Settings  Help
zenoss@zen241:/usr/local/zenoss/zenoss/local/jane/ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/skin
s/ZenPacks.skills1st.bridge> snmpwalk -v 1 -c public  switch.skills-1st.co.uk .1.3.6.1.2.1.17.4.3.1.2
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.192 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.193 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.194 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.195 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.196 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.197 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.198 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.199 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.200 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.201 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.202 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.203 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.204 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.205 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.206 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.207 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.208 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.209 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.210 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.211 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.212 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.213 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.214 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.215 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.4.193.156.144.216 = INTEGER: 40
SNMPv2-SMI::mib-2.17.4.3.1.2.0.12.65.157.211.129 = INTEGER: 13
SNMPv2-SMI::mib-2.17.4.3.1.2.0.14.53.100.114.167 = INTEGER: 13
SNMPv2-SMI::mib-2.17.4.3.1.2.0.17.37.128.28.79 = INTEGER: 13
SNMPv2-SMI::mib-2.17.4.3.1.2.1.0.12.0.0.0 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.0.12.204.204.204 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.0.12.204.204.205 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.0.12.221.221.221 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.0 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.1 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.2 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.3 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.4 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.5 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.6 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.7 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.8 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.9 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.10 = INTEGER: 0
SNMPv2-SMI::mib-2.17.4.3.1.2.1.128.194.0.0.11 = INTEGER: 0
```

Figure 24: Results from performing snmpwalk for the Port values of the Port Forwarding table of the BRIDGE MIB

The Port values are also indexed by the same representation of the MAC address in decimal. The only “real” values shown in Figure 24 are for port 13 (as only one port actually has anything connected to it). The other values of 0 and 40 are for internal and management addresses.

```
jane@zen241:~ - Shell - Konsole <3>
Session  Edit  View  Bookmarks  Settings  Help
zenoss@zen241:/usr/local/zenoss/zenoss/local/jane/ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/skin
s/ZenPacks.skills1st.bridge> snmpwalk -v 1 -c public  switch.skills-1st.co.uk .1.3.6.1.2.1.17.4.3.1.3
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.192 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.193 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.194 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.195 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.196 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.197 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.198 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.199 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.200 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.201 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.202 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.203 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.204 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.205 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.206 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.207 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.208 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.209 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.210 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.211 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.212 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.213 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.214 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.215 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.4.193.156.144.216 = INTEGER: 4
SNMPv2-SMI::mib-2.17.4.3.1.3.0.12.65.157.211.129 = INTEGER: 3
SNMPv2-SMI::mib-2.17.4.3.1.3.0.17.37.128.28.79 = INTEGER: 3
SNMPv2-SMI::mib-2.17.4.3.1.3.0.22.212.93.8.253 = INTEGER: 3
SNMPv2-SMI::mib-2.17.4.3.1.3.1.0.12.0.0.0 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.0.12.204.204.204 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.0.12.204.204.205 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.0.12.221.221.221 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.0 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.1 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.2 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.3 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.4 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.5 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.6 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.7 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.8 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.9 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.10 = INTEGER: 5
SNMPv2-SMI::mib-2.17.4.3.1.3.1.128.194.0.0.11 = INTEGER: 5
```

Figure 25: Results from performing snmpwalk for the PortStatus values of the Port Forwarding table of the BRIDGE MIB

The same indexing technique is used again. A PortStatus of 3 represents a "real" *learned* value. A value of 4 represents *self* addresses and a value of 5 represents *mgmt* addresses.

4.3.3 Creating the ZenPack

The ZenPack is created from the GUI as discussed in section 2.1. The name of the ZenPack is **ZenPacks.skills1st.bridge**. A dependency of Zenoss >= 2.2 was imposed.

Once the directory hierarchy is created, rather than working under $ZENHOME/ZenPacks, the whole ZenPack directory hierarchy is moved to $ZENHOME/local jane to prevent accidental deletion:

```
cp -r $ZENHOME/ZenPacks/ZenPacks.skills1st.bridge $ZENHOME/local/jane
```

The ZenPack is then “reinstalled” with the zenpack –link --install command:

```
zenpack --link --install $ZENHOME/local/jane/ZenPacks.skills1st.bridge
```

At this stage, the ZenPack can be modified either using Development mode (ie from GUI menus), or by modifying files in the directory hierarchy (Source mode); a combination of both is perfectly acceptable and both will follow the redirection link.

4.3.4 Adding elements to the ZenPack using Development mode

The first thing to do is to create the new device class, *BridgeMIB*, as a subclass of */Devices/Network/Switch*. This is simply achieved with the GUI using the drop-down *Add New Organizer* menu, as shown in Figure 22. Once created, this device class can be added to a ZenPack with the drop-down *Add To ZenPack* menu – you will be prompted for the ZenPack to which it is to be added.

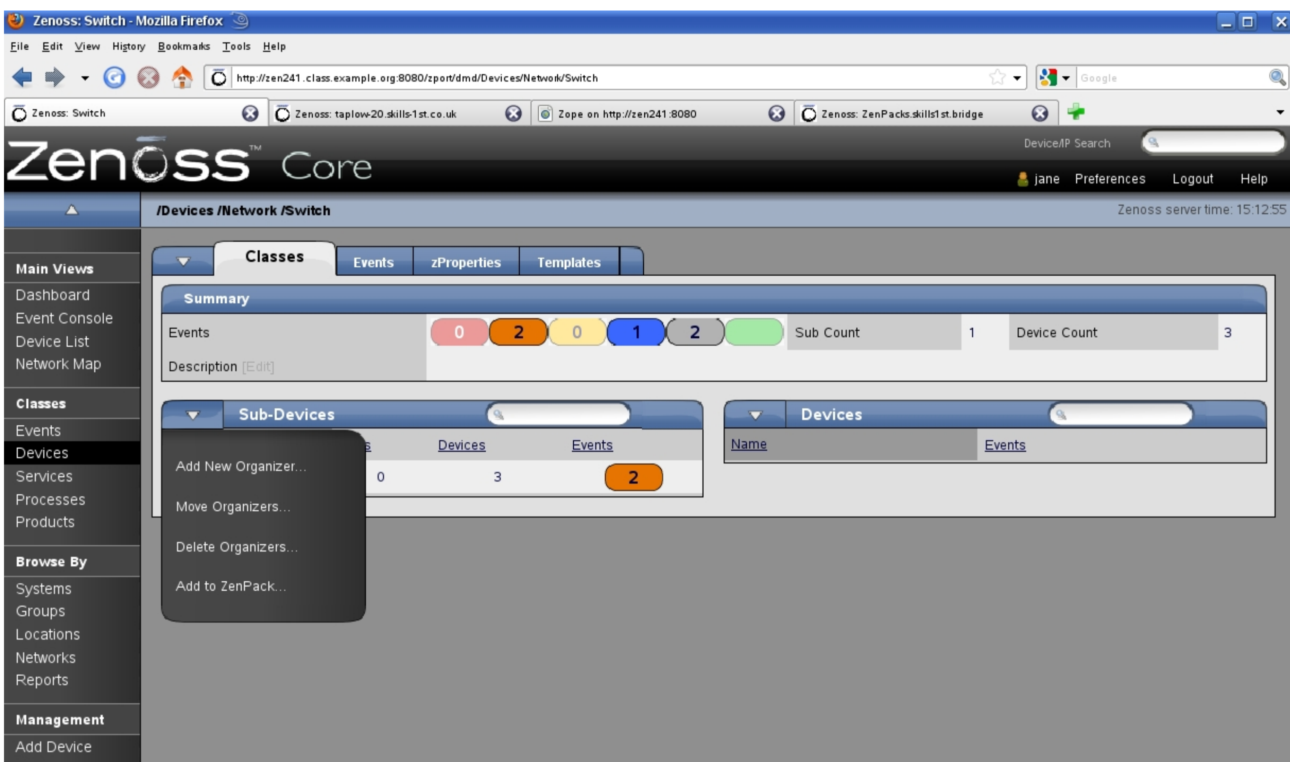

Figure 26: Adding a device class to a ZenPack

If the BridgeMIB device class is subsequently modified, it should be re-added in the same way, overwriting the previous version.

It is useful to have relevant MIBs loaded and included as part of any ZenPack. The BRIDGE MIB and the standard SNMP MIBs had already been loaded (using the left-hand MIBs menu). To include these in a ZenPack, simply select the relevant MIBs and use the drop-down *Add to ZenPack* menu.

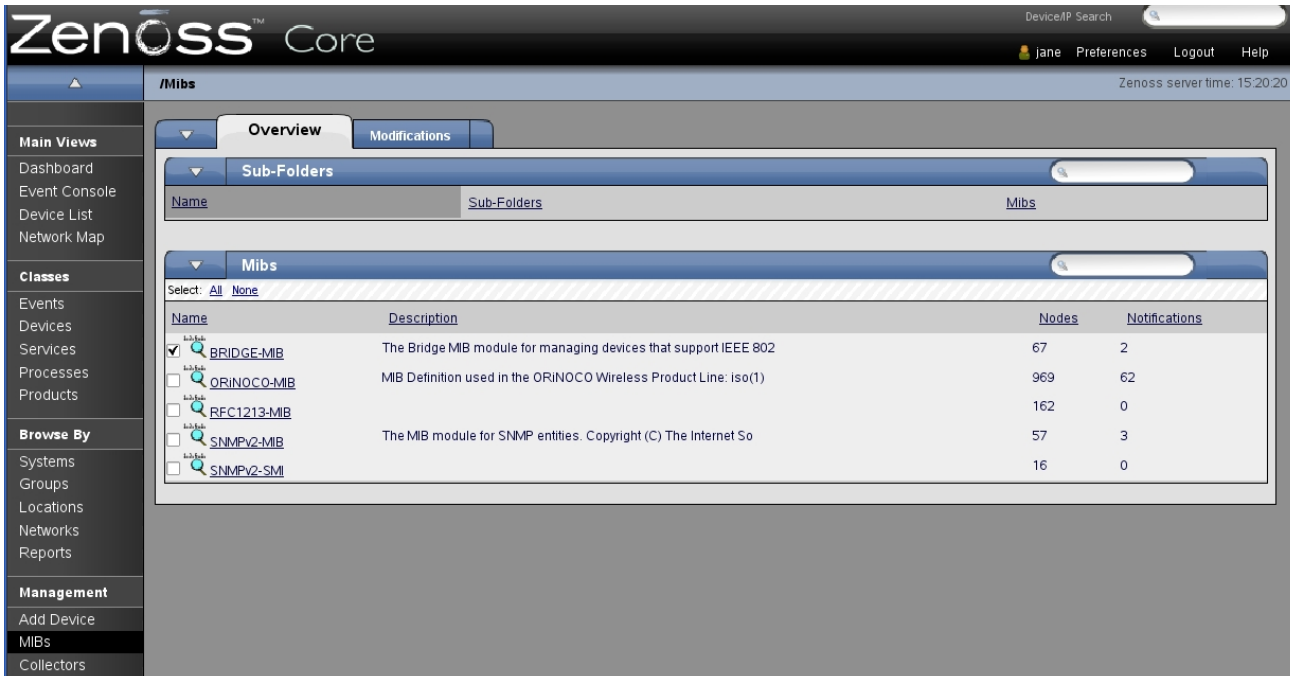

Figure 27: Use the drop-down Mibs table menu to select Add to ZenPack

The contents of a ZenPack can be seen at any stage by using the left-hand *Settings* menu, choosing the *ZenPacks* tab, and selecting the relevant ZenPack.

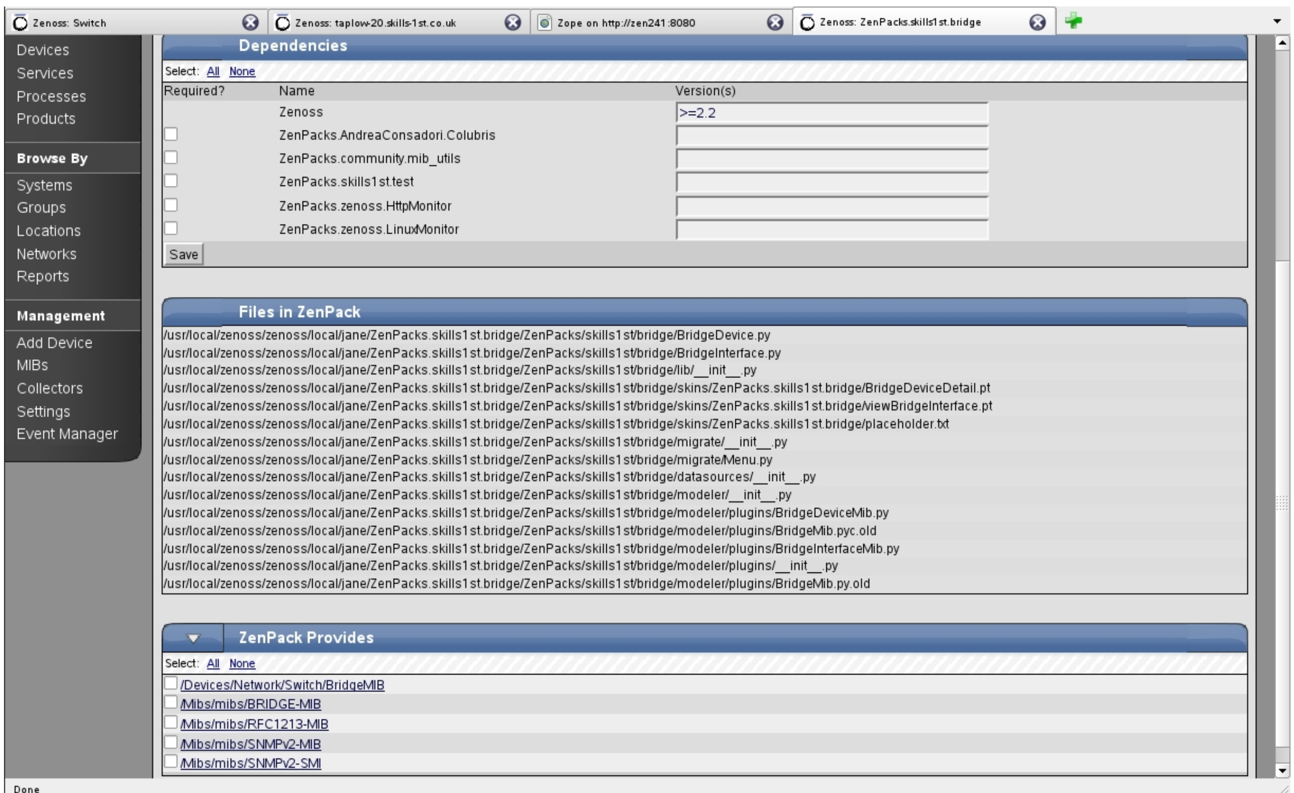

Figure 28: Inspecting the contents of the ZenPacks.skills1st.bridge ZenPack

4.3.5 Creating the object class files

Two object class files are needed; one will represent the device itself (BridgeDevice) and one will represent an interface on the device (BridgeInterface). The two are linked by a matching pair of relationships. Both files must be in the ZenPack base directory.

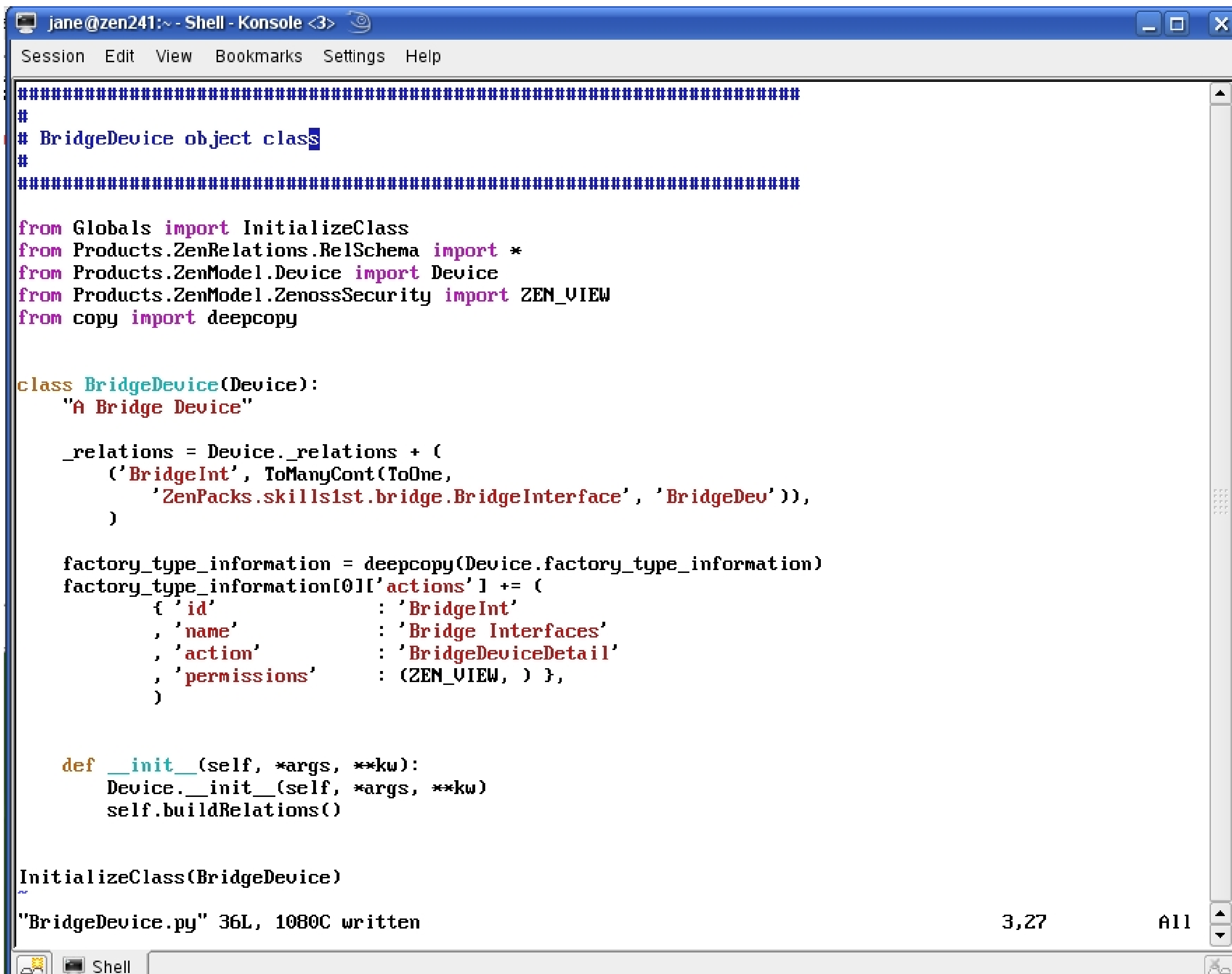

Figure 29: BridgeDevice.py object class file in ZenPack base directory

This is a very simple object class file as it does not define any unique field attributes, only a relationship and a skins file.

The BridgeDevice class inherits from the base Device class:

```
class BridgeDevice(Device):
```

The relationship stanza adopts all existing relations for the base Device class and adds on a relationship called *BridgeInt* of type *ToManyCont*, with the device object class defined in *ZenPacks.skills1st.bridge.BridgeInterface* (which corresponds to the file under the ZenPack base directory called *BridgeInterface.py*).

```
_relations = Device._relations + (
     'BridgeInt', ToManyCont(ToOne,
        'ZenPacks.skills1st.bridge.BridgeInterface', 'BridgeDev')),
        )
```

The BridgeDevice object class will have all the standard menu options for the base Device class and will also have an extra tab whose id is *BridgeInt*; whose tab label will be *Bridge Interfaces*; whose page layout will be specified by the file *BridgeDeviceDetail*.pt under the skins/ZenPacks.skills1st.bridge subdirectory of the ZenPack base directory. Access permissions to use this tab is the standard-supplied *ZEN_VIEW*.

```
factory_type_information = deepcopy(Device.factory_type_information)
factory_type_information[0]['actions'] += (
        { 'id'              : 'BridgeInt'
        , 'name'            : 'Bridge Interfaces'
        , 'action'          : 'BridgeDeviceDetail'
        , 'permissions'     : (ZEN_VIEW, ) },
        )
```

A Python function *__init__* is defined for the BridgeDevice object class which will initialize the object and create relationships.

```
def __init__(self, *args, **kw):
        Device.__init__(self, *args, **kw)
        self.buildRelations()
```

The last line delivers the new object.

```
InitializeClass(BridgeDevice)
```

The BridgeInterface.py object class file is more interesting as some unique fields are defined in addition to a relationship and a skins file. Several extra functions are also defined which will be used in the skins files.

```
jane@zen241:~ - Shell - Konsole <3>
Session  Edit  View  Bookmarks  Settings  Help
######################################################################
#
# BridgeInterface object class
#
######################################################################

__doc__="""BridgeInt

BridgeInt is a component of a Bridge Device

$Id: $"""

__version__ = "$Revision: $"[11:-2]

from Globals import DTMLFile
from Globals import InitializeClass

from Products.ZenRelations.RelSchema import *
from Products.ZenModel.ZenossSecurity import ZEN_VIEW, ZEN_CHANGE_SETTINGS

from Products.ZenModel.DeviceComponent import DeviceComponent
from Products.ZenModel.ManagedEntity import ManagedEntity


class BridgeInterface(DeviceComponent, ManagedEntity):
    """Bridge Interface object"""

    event_key = portal_type = meta_type = 'BridgeInterface'

    #***************Custom data Variables here from modeling************************

    RemoteAddress = '00:00:00:00:00:00'
    Port = '-1'
    PortIfIndex = 2
    PortStatus = '4'

    #***************END CUSTOM VARIABLES *****************************


    #**************  Those should match this list below *******************
    _properties = (
        {'id':'RemoteAddress', 'type':'string', 'mode':''},
        {'id':'Port', 'type':'string', 'mode':''},
        {'id':'PortIfIndex', 'type':'int', 'mode':''},
        {'id':'PortStatus', 'type':'string', 'mode':''}
        )
    #*****************
"BridgeInterface.py" [readonly] 135 lines --0%--                    1,1           Top
```

Figure 30: BridgeInterface.py object class file - first part with unique field definitions

The BridgeInterface class is defined as a DeviceComponent and a ManagedEntity.

```
class BridgeInterface(DeviceComponent, ManagedEntity):
```

There are 4 unique fields defined for a BridgeInterface object:

- RemoteAddress
- Port
- PortIfIndex
- PortStatus

The data types (such as *string* or *int*) must be specified and the mode of read or write ('w') may be specified.

The middle part of BridgeInterface.py defines the relationship with BridgeDevice and three web pages associated with this object.

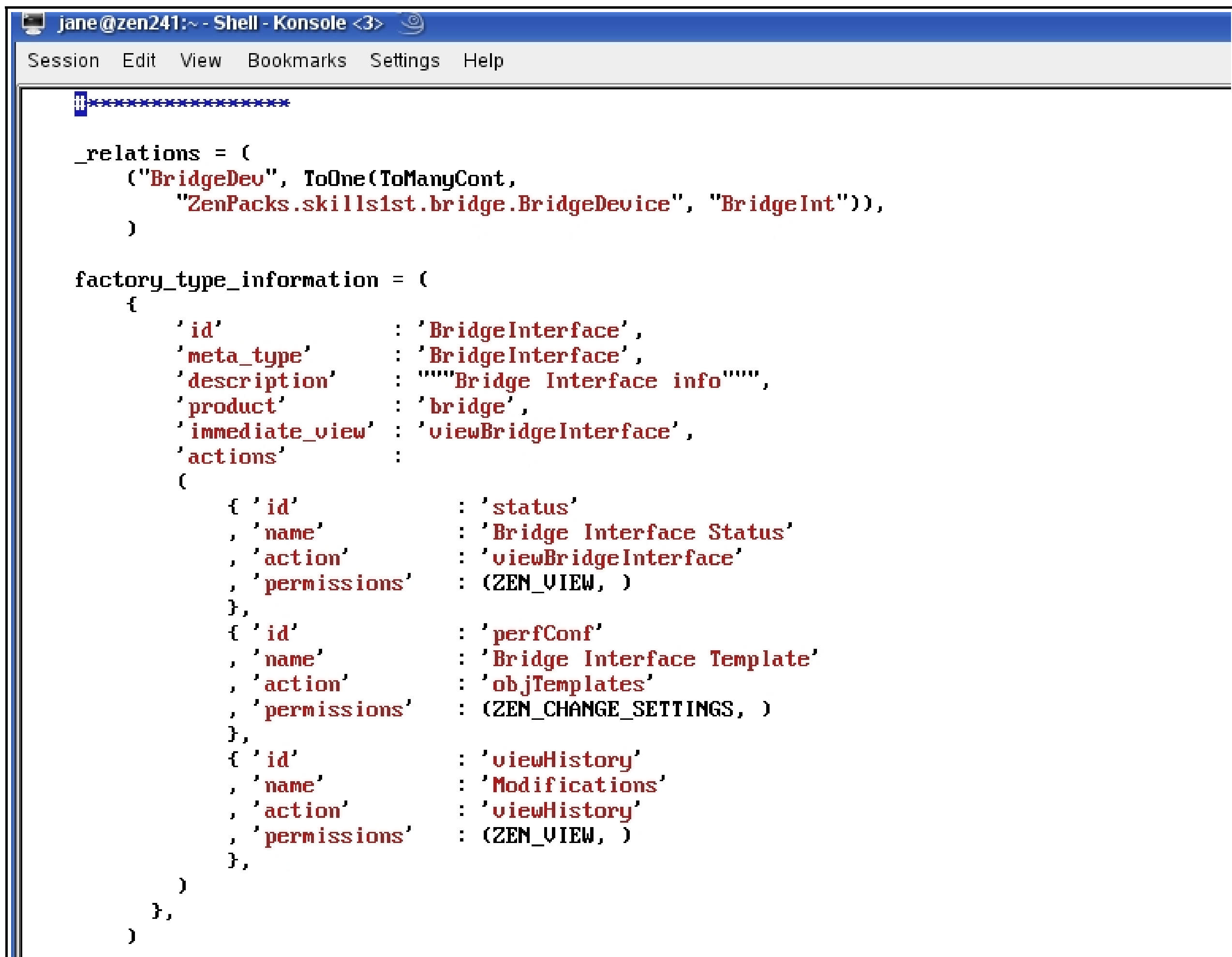

Figure 31: BridgeInterface.py showing relations and web pages

The first skins file that is referenced, *viewBridgeInterface*, is part of this ZenPack and thus is to be found in the skins/ZenPacks.skills1st.bridge subdirectory; the other two files, *objTemplates* and *viewHistory* are standard pages provided by Zenoss and these .pt files are found in $ZENHOME/Products/ZenModel/skins/zenmodel.

Note the product line in the factory_type_information:

```
'product'          : 'bridge'
```

The value of *'bridge'* denotes the last part of the ZenPack name (and hence the directory hierarchy) ie. ZenPacks.skills1st.*bridge*. Unlike the BridgeDevice definitions of skins files, nothing is inherited from the base Device object. The resulting web page with its three tabs can be seen in Figure 32.

Figure 32: The web page having drilled into the interface of a switch port - note the 3 tabs

An object class definition file can specify not only object attributes but also methods for the object; these are coded as function definitions in Python. The methods can then be used in skins files to augment the data that is displayed.

 © Skills 1st Ltd

```
jane@zen241:...s/skills1st/bridge - Shell - Konsole
Session  Edit  View  Bookmarks  Settings  Help

    def viewName(self):
        if self.RemoteAddress == '00:00:00:00:00:00' \
            or self.Port == '-1':
                return "Unknown"
        else:
                return str( self.Port ) + "/" + self.RemoteAddress

#    name = primarySortKey = viewName
    name = viewName

    def primarySortKey(self):
        """sort on Port status"""
        return self.PortStatus

    def device(self):
        return self.BridgeDev()

    def getIpRemoteAddress(self):
        dmd=self.dmd
        devmac=self.RemoteAddress
        IpAddress=[]
        Ips=dmd.ZenLinkManager.layer2_catalog(macaddress=devmac)
        for i in Ips:
            IpAddress=IpAddress + [i.getObject().manageIp]
        return IpAddress

    def getIpRemoteIfDesc(self):
        dmd=self.dmd
        devmac=str(self.RemoteAddress)
        IfDesc=[]
        Ips=dmd.ZenLinkManager.layer2_catalog(macaddress=devmac)
        for i in Ips:
            IfDesc=IfDesc + [i.getObject().id]
        return IfDesc

    def getIpRemoteHostname(self):
        dmd=self.dmd
        find = dmd.Devices.findDevice
        devmac=str(self.RemoteAddress)
        Hostname=[]
        Ips=dmd.ZenLinkManager.layer2_catalog(macaddress=devmac)
        for i in Ips:
            Hostname=Hostname + [find(i.getObject().manageIp).id]
        return Hostname


    #THIS FUNCTION IS REQUIRED LEAVE IT BE IF NO RRD INFO IS PRESENT
    def getRRDNames(self):
        return []

InitializeClass(BridgeInterface)
"BridgeInterface.py" [readonly] 135 lines --100%--                135,1        Bot
Shell
```

Figure 33: BridgeInterface.py part 3 showing functions defined for this object class

The first function, viewName, returns either the string "Unknown" or a string that concatenates the Port number with a "/" and the RemoteAddress. For example, 13/00:11:25:80:1C:4F .

```
def viewName(self):
    if self.RemoteAddress == '00:00:00:00:00:00' \
        or self.Port == '-1':
            return "Unknown"
    else:
            return str( self.Port ) + "/" + self.RemoteAddress

name = viewName
```

There are three similar functions that use the MAC address delivered by the RemoteAddress field and then search the Zope database for devices that have a matching MAC address, delivering from the database the corresponding interface IP address, Interface description or Hostname. A Python list is returned (denoted by square brackets [ ]) as there may be more than one matching value.

```
def getIpRemoteAddress(self):
    dmd=self.dmd
    devmac=self.RemoteAddress
    IpAddress=[]
    Ips=dmd.ZenLinkManager.layer2_catalog(macaddress=devmac)
    for i in Ips:
        IpAddress=IpAddress + [i.getObject().manageIp]
    return IpAddress
```

The getObject() method is called with the required attribute of the device – *manageIp* in the case of the getIpRemoteAddress function.

One way to find what attributes of a device are available, is to use the Zenoss *zendmd* utility and run a small series of Python commands:

```
zendmd
>>> dev=find('switch.skills-1st.co.uk')
>>> for key,value in dev.__dict__.items():
...   print key,value
...
```

Note that >>> is the zendmd prompt and . . . indicates that a new level of indentation is required. A blank line ends the code and runs the Python, delivering results similar to those shown in Figure 34.

jane@zen241:~ - Shell - Konsole <2>

Session Edit View Bookmarks Settings Help

```
...
_lastChange 1249382930.2
snmpContact andrew.findlay@skills-1st.co.uk
preMWProductionState 1000
_snmpLastCollection 1249414843.41
rackSlot 0
id switch.skills-1st.co.uk
maintenanceWindows <ToManyContRelationship at maintenanceWindows>
adminRoles <ToManyContRelationship at adminRoles>
__primary_parent__ <ToManyContRelationship at devices>
comments
monitors <ToManyRelationship at monitors>
priority 3
_temp_device False
systems <ToManyRelationship at systems>
_objects ({'meta_type': 'ToManyRelationship', 'id': 'dependencies'}, {'meta_type': 'ToManyRelationship', '
id': 'dependents'}, {'meta_type': 'ToOneRelationship', 'id': 'deviceClass'}, {'meta_type': 'ToOneRelations
hip', 'id': 'perfServer'}, {'meta_type': 'ToOneRelationship', 'id': 'location'}, {'meta_type': 'ToManyRela
tionship', 'id': 'systems'}, {'meta_type': 'ToManyRelationship', 'id': 'groups'}, {'meta_type': 'ToManyCon
tRelationship', 'id': 'maintenanceWindows'}, {'meta_type': 'ToManyContRelationship', 'id': 'adminRoles'},
{'meta_type': 'ToManyContRelationship', 'id': 'userCommands'}, {'meta_type': 'ToManyRelationship', 'id': '
monitors'}, {'meta_type': 'ToManyContRelationship', 'id': 'BridgeInt'}, {'meta_type': 'Software', 'id': 'o
s'}, {'meta_type': 'DeviceHW', 'id': 'hw'})
location <ToOneRelationship at location>
_lastPollSnmpUpTime <Products.ZenModel.ZenStatus.ZenStatus object at 0x96d442c>
snmpOid .1.3.6.1.4.1.9.1.217
hw <DeviceHW at hw>
snmpDescr Cisco Internetwork Operating System Software
IOS (tm) C2900XL Software (C2900XL-C3H2S-M), Version 12.0(5.1)XP, MAINTENANCE INTERIM SOFTWARE
Copyright (c) 1986-1999 by cisco Systems, Inc.
Compiled Fri 10-Dec-99 10:37 by cchang
dependencies <ToManyRelationship at dependencies>
groups <ToManyRelationship at groups>
perfServer <ToOneRelationship at perfServer>
deviceClass <ToOneRelationship at deviceClass>
snmpSysName switch.skills-1st.co.uk
productionState 1000
zCollectorPlugins ['zenoss.snmp.NewDeviceMap', 'zenoss.snmp.DeviceMap', 'zenoss.snmp.InterfaceMap', 'zenos
s.snmp.RouteMap', 'BridgeInterfaceMib', 'BridgeDeviceMib']
manageIp 10.0.0.253
BridgeInt <ToManyContRelationship at BridgeInt>
_properties ({'type': 'string', 'id': 'snmpindex', 'mode': 'w'}, {'type': 'boolean', 'id': 'monitor', 'mod
e': 'w'}, {'type': 'string', 'id': 'manageIp', 'mode': 'w'}, {'select_variable': 'getProdStateConversions'
, 'setter': 'setProdState', 'type': 'keyedselection', 'id': 'productionState', 'mode': 'w'}, {'select_vari
able': 'getProdStateConversions', 'setter': 'setProdState', 'type': 'keyedselection', 'id': 'preMWProducti
onState', 'mode': 'w'}, {'type': 'string', 'id': 'snmpAgent', 'mode': 'w'}, {'type': 'string', 'id': 'snmp
```

Figure 34: Output of zendmd commands to print attributes of the device switch.skills-1st.co.uk

The getIpRemoteIfDesc function delivers the interface description of the remote device interface:

```
def getIpRemoteIfDesc(self):
    dmd=self.dmd
    devmac=str(self.RemoteAddress)
    IfDesc=[]
    Ips=dmd.ZenLinkManager.layer2_catalog(macaddress=devmac)
    for i in Ips:
        IfDesc=IfDesc + [i.getObject().id]
    return IfDesc
```

Note that the i.getObject().id value relates to a device interface, not the device itself, so the id attribute is the interface description, not the hostname of the device.

To get the hostname associated with a remote IP address in the Zope database, an extra twist is required:

```
def getIpRemoteHostname(self):
    dmd=self.dmd
    find = dmd.Devices.findDevice
    devmac=str(self.RemoteAddress)
    Hostname=[]
    Ips=dmd.ZenLinkManager.layer2_catalog(macaddress=devmac)
    for i in Ips:
        Hostname=Hostname + [find(i.getObject().manageIp).id]
    return Hostname
```

The remote interface IP address is delivered by i.getObject().manageIp. The find function then takes that IP address as a parameter and looks for a device with the same manageIp address and delivers the id attribute of the device – that is, the device hostname.

Ultimately, we want to have performance graphs related to the interfaces of a switch port so the final function is required:

```
def getRRDNames(self):
    return []
```

4.3.6 Creating the modeler plugin files

This ZenPack has two modeler plugin files, residing under the base ZenPack directory under the modeler/plugins subdirectory hierarchy. They are:

- BridgeInterfaceMib.py gets port data for each switch port
- BridgeDeviceMib.py gets scalar data for the switch device

These names can be anything but should obviously be relevant. The only place where these names appear is when a device or device class has its Collector Plugins configured from the Zenoss GUI. The purpose of a modeler plugin is to **map** collected data into the attributes of Zenoss objects.

jane@zen241:~ - Shell - Konsole <3>

Session Edit View Bookmarks Settings Help

```
##########################################################################
#
# BridgeInterfaceMib modeler plugin
#
##########################################################################

__doc__="""BridgeInterfaceMib

BridgeInterfaceMib maps interfaces on a switch supporting the Bridge MIB

$Id: $"""

__version__ = '$Revision: $'[11:-2]

from Products.DataCollector.plugins.CollectorPlugin import SnmpPlugin, GetTableMap, GetMap
from Products.DataCollector.plugins.DataMaps import ObjectMap

class BridgeInterfaceMib(SnmpPlugin):

    relname = "BridgeInt"
    modname = "ZenPacks.skills1st.bridge.BridgeInterface"
#    compname not needed as BridgeInt is a relationship on object class BridgeDevice
#    which is a direct child of Device"
#    compname = ""

    basecolumns = {
                 '.1': 'BasePort',
                 '.2': 'BasePortIfIndex',
               }

    portcolumns = {
                 '.1': 'RemoteAddress',
                 '.2': 'Port',
                 '.3': 'PortStatus',
               }

# snmpGetTableMaps gets tabular data

    snmpGetTableMaps = (
        # Physical Port Forwarding Table
        GetTableMap('dot1dBasePortEntry', '.1.3.6.1.2.1.17.1.4.1', basecolumns),

        # Physical Port Forwarding Table
        GetTableMap('dot1dTpFdbEntry', '.1.3.6.1.2.1.17.4.3.1', portcolumns),
    )
                                                                1,1           Top
```

Figure 35: BridgeInterfaceMib modeler plugin (part 1) with SNMP data to be collected

The first part of the BridgeInterfaceMib modeler plugin code imports some standard Zenoss utilities for getting SNMP information and formatting it.

```
from Products.DataCollector.plugins.CollectorPlugin import SnmpPlugin, GetTableMap, GetMap
from Products.DataCollector.plugins.DataMaps import ObjectMap
```

Note that the modeler plugin, BridgeInterfaceMib, is itself defined as an object class which derives from the standard SnmpPlugin modeler. The modeler must be activated for a device or device class (from the *More -> Collector Plugins* menu) – it cannot be directly activated for a device component such as a port on a switch. Hence, the *relname* and *modname* directives specify that the data is to be applied to a

relationship of the device, the component object class being specified by the modname line.

```
class BridgeInterfaceMib(SnmpPlugin):

    relname = "BridgeInt"
    modname = "ZenPacks.skills1st.bridge.BridgeInterface"
```

In other words, the modeler is applied to a device of object class BridgeDevice but the data will be mapped to the contained relationship called BridgeInt whose data attributes are specified by ZenPacks.skills1st.bridge.BridgeInterface; this comes down to populating the unique RemoteAddress, Port, PortIfIndex and PortStatus attributes.

The next part of the modeler plugin specifies SNMP data tables and ObjectIDs (OIDs) to collect.

```
    basecolumns = {
                 '.1': 'BasePort',
                 '.2': 'BasePortIfIndex',
               }

    portcolumns = {
                 '.1': 'RemoteAddress',
                 '.2': 'Port',
                 '.3': 'PortStatus',
               }

# snmpGetTableMaps gets tabular data

    snmpGetTableMaps = (
        # Physical Port Forwarding Table
        GetTableMap('dot1dBasePortEntry', '.1.3.6.1.2.1.17.1.4.1',
basecolumns),

        # Physical Port Forwarding Table
        GetTableMap('dot1dTpFdbEntry', '.1.3.6.1.2.1.17.4.3.1',
portcolumns),
    )
```

The GetTableMap standard Zenoss function takes three parameters:

- A table name you'll be using later (this can be anything but it is helpful if it matches the name of the SNMP table)
- The OID of the SNMP table
- A dictionary of "OID-endings" and column names (OID-endings being the keys, used later)

If there are only one or two OIDs required, it is perfectly possible to code them directly as part of GetTableMap. It is also possible to specify the OID-ending as more than the last digit. For example, the following code has the same effect as the first GetTableMap stanza above.

```
GetTableMap('dot1dBasePortEntry', '.1.3.6.1.2.1.17.1.4',
    {'1.1': 'BasePort',
     '1.2': 'BasePortIfIndex',
    }
```

```
    ),
```

It is usually clearer and more convenient to specify the dictionary of "OID-endings" and column names separately as shown above with basecolumns.

The snmpGetTableMaps function can get one or more SNMP tables of data.

The only mandatory function required in a modeler plugin is the *process()* function.

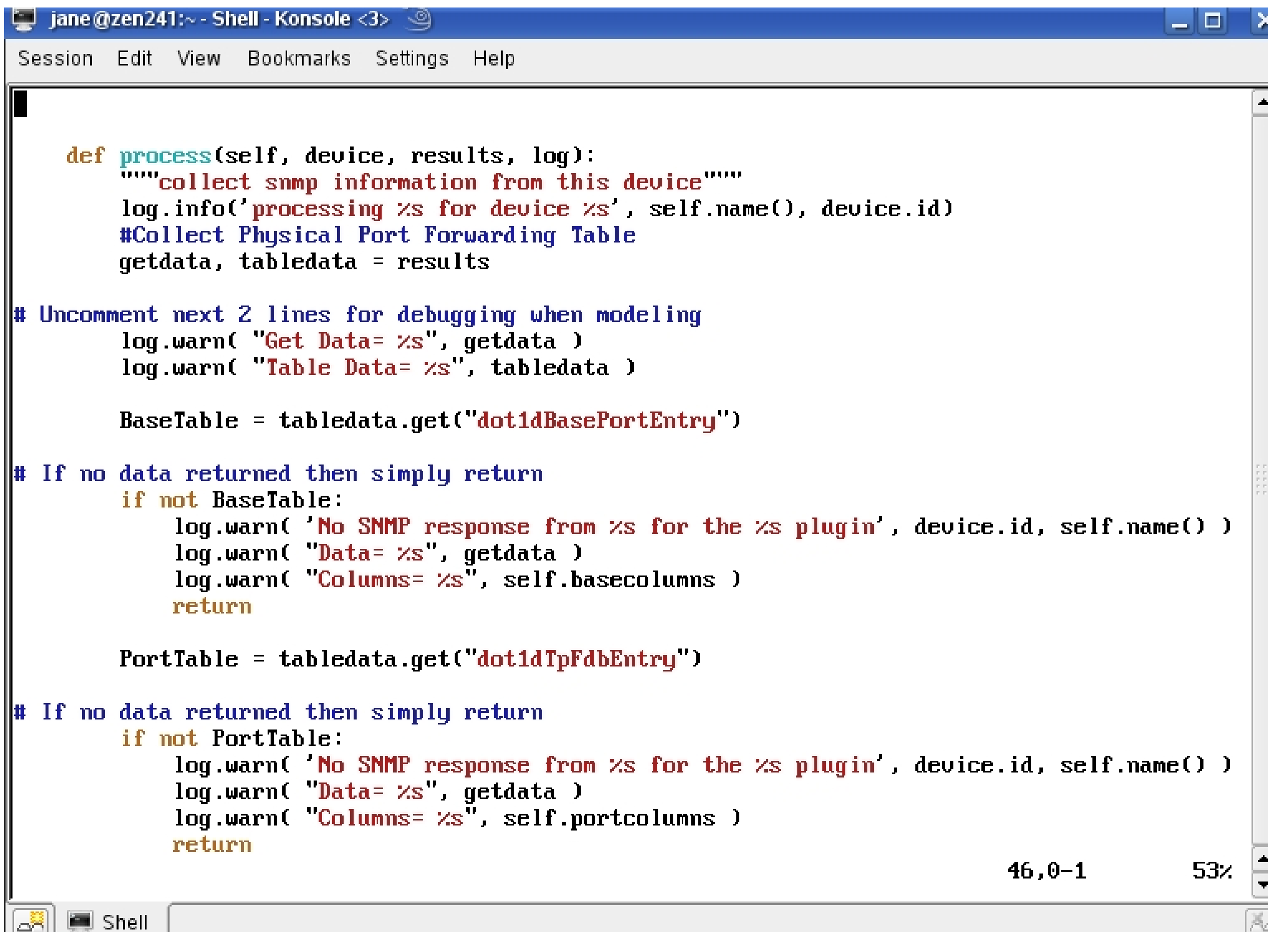

Figure 36: BridgeInterfaceMib modeler plugin (part 2) showing data collection and error checking

The part that actually gets the data is the line:

```
getdata, tabledata = results
```

Scalar data is populated into *getdata*; table data is populated into *tabledata*. Debugging can be provided using *log* statements with different severities such as log.*info* and log.*warn*.

```
log.info('processing %s for device %s', self.name(), device.id)
```

Remember that snmpGetTableMaps retrieves two tables of data into the variables dot1dBasePortEntry and dot1dTpFdbEntry. The second half of Figure 36 checks that SNMP data was actually retrieved (as the device may, for example, have been down

on a modeler cycle). If either table is not populated then logging is produced and the process function simply returns.

The last part of the modeler plugin code creates a relationship mapping that will contain entries for each object that represents a port on the device.

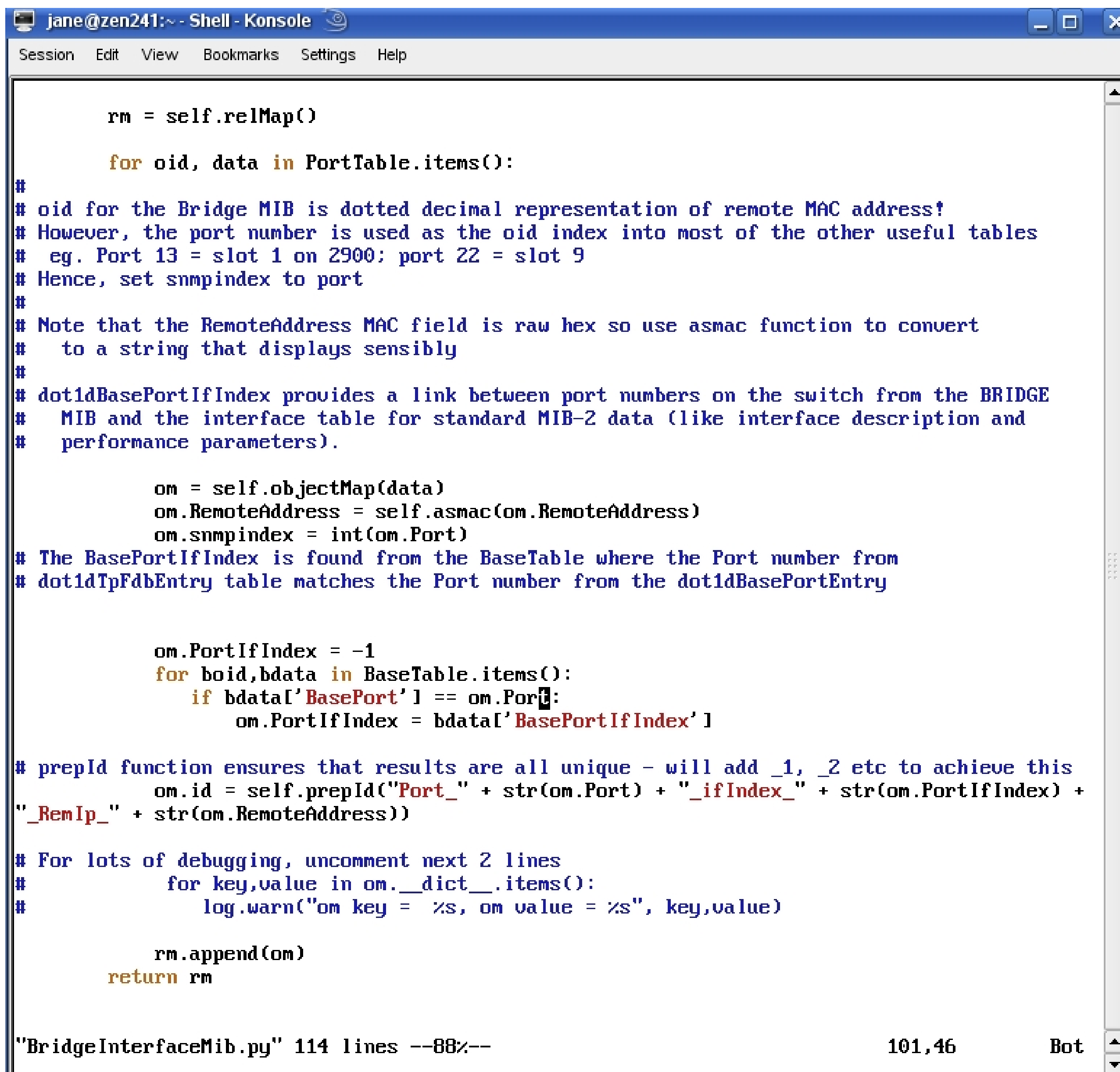

Figure 37: BridgeInterfaceMib modeler plugin (part 3) mapping and modifying SNMP data onto objects

Remember that the GetTableMap delivers a table (strictly a Python dictionary). The two fields of the dictionary are the OID and the data; the data itself is also a dictionary containing column names and values. To see what is actually delivered, make sure that the following lines are uncommented and then model a switch device from the *Manage* -> *Model* device menu.

```
# Uncomment next 2 lines for debugging when modeling
        log.warn( "Get Data= %s", getdata )
```

```
            log.warn( "Table Data= %s", tabledata )
```

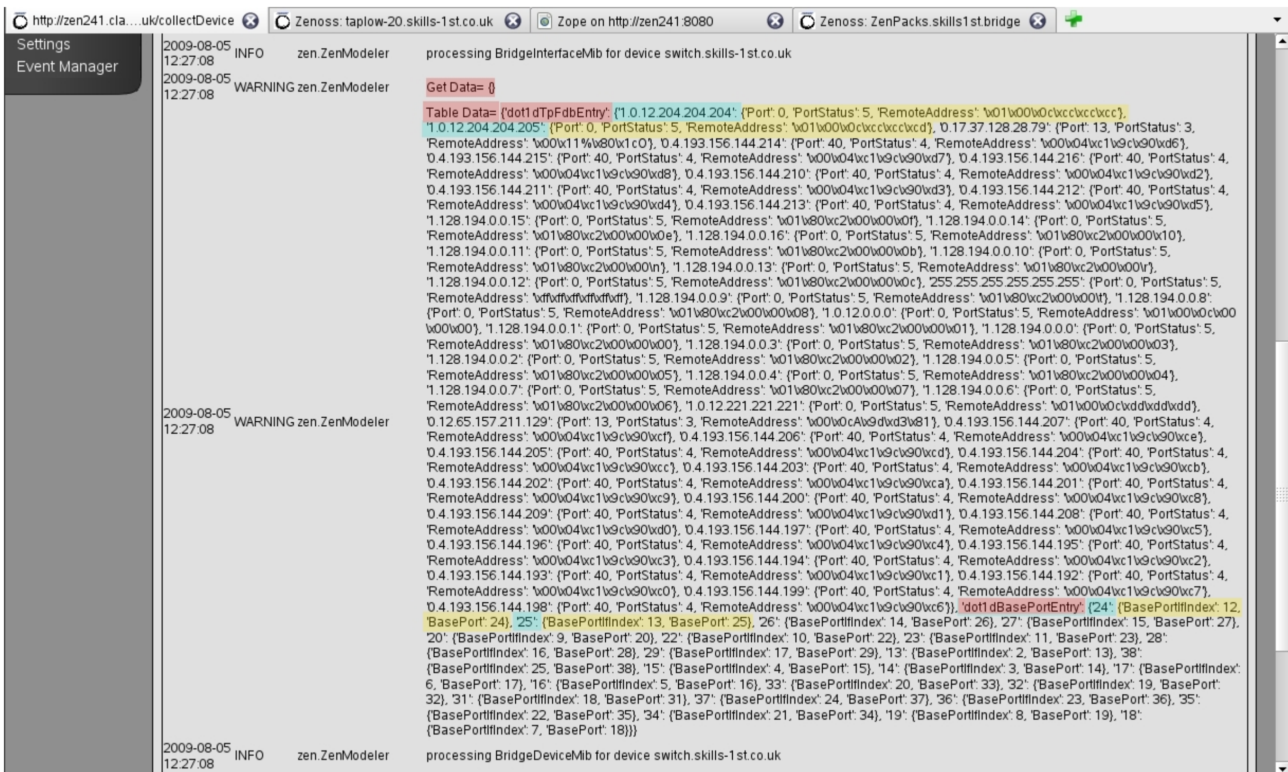

Figure 38: Debug output for BridgeInterfaceMib modeler plugin

The BridgeInterfaceMib modeler plugin doesn't, in fact, get scalar data, so the the getdata dictionary is empty (ie. {}). snmpGetTableMaps delivers two tables (Python dictionaries) – dot1dTpFdbEntry and dot1dBasePortEntry; these are all shown highlighted in red in Figure 38. Each of dot1dTpFdbEntry and dot1dBasePortEntry comprises a dictionary with OID and data components. The first few OID values are highlighted in blue for each table. The data component is itself a dictionary with column names and values; these are highlighted in yellow.

So, the lines:

```
for oid, data in PortTable.items():
  om=self.objectMap(data)
```

cycles through each of the OID, data sets of values in the PortTable, mapping the data values to the attributes of the BridgeInterface object; Port, PortStatus and RemoteAddress.

Note in Figure 38 that the MAC address is in hex format. To display this for users, it needs converting to a string-type representation so the delivered value of the RemoteAddress is converted using the Python *asmac* function, replacing the RemoteAddress value on the object.

```
om.RemoteAddress = self.asmac(om.RemoteAddress)
```

The ZenPack only defined four unique attributes for the BridgeInterface object in the object class file BridgeInterface.py:

- RemoteAddress
- Port
- PortIfIndex
- PortStatus

However, it also inherited attributes as a DeviceComponent and ManagedEntity and thus has other attributes, including:

- id
- snmpindex

The *id* should be a unique and meaningful identifier. *snmpindex* is used when performance data is configured using Zenoss templates and provides the instance to collect for any given SNMP OID. Most of the useful SNMP data to do with switch ports is actually indexed using the value of the port number (remember for the test Catalyst 2900 switch, the values of Port representing real interfaces run from 13 to 38 – you can see the values for a real active port in Figure 38 right in the middle, opposite WARNING zen.ZenModeler). Hence, the snmpindex attribute is set to the Port value, having first converted the raw data to an integer type.

```
om.snmpindex = int(om.Port)
```

Note that many modeler plugins use the OID value from the tabledata as the snmpindex but this is only useful if that OID does actually represent a useful SNMP index. The OID value that we have delivered (highlighted in blue in Figure 38) is the decimal representation of a MAC address and is not useful as an instance for collecting performance information. More on this topic later.

A switch discovered by Zenoss will automatically gather information on each of the ports, using information from the MIB-2 MIB. This doesn't provide much port-level information but it does provide some. The interfaces are indexed using the interfaces table of MIB-2.

Figure 39: Standard OS tab for switch device with MIB-2 interfaces

Drilling in to an interface results in both tabular information and performance graphs for bound templates.

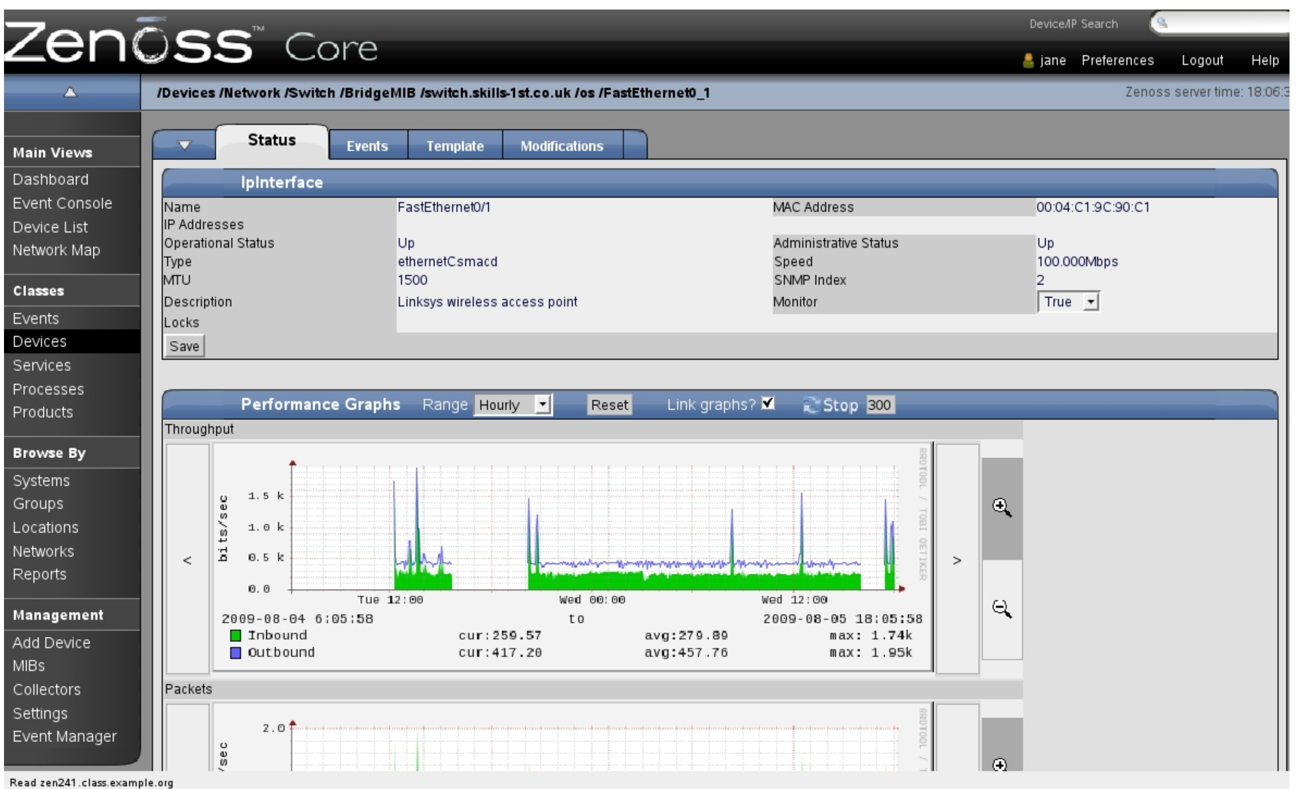

Figure 40: Tabular information and performance graphs for a switch interface from MIB-2

Note in the table at the top of Figure 40 that the SNMP Index for this interface is given as *2*. It is this index number that is delivered by the BRIDGE MIB to coordinate MIB-2 interface information with BRIDGE MIB information. The OID is the BasePortIfIndex from the dot1dBasePortEntry table (.1.3.6.1.2.1.17.1.4.1.2). . 1.3.6.1.2.1.17.1.4.1.1 (BasePort from the same table) will be the same as the Port OID from the forwarding table (.1.3.6.1.2.1.17.4.3.1.2). The following code delivers the PortIfIndex attribute to the BridgeInterface object, if a valid PortIfIndex exists (it won't for internal and management ports); otherwise PortIfIndex will be -1.

```
# The BasePortIfIndex is found from the BaseTable where  Port number from
# dot1dTpFdbEntry table matches the Port number from the dot1dBasePortEntry

           om.PortIfIndex = -1
            for boid,bdata in BaseTable.items():
               if bdata['BasePort'] == om.Port:
                   om.PortIfIndex = bdata['BasePortIfIndex']
```

The last attribute to populate is the unique *id* attribute. This is constructed by concatenating the string "Port_"with the Port number, followed by the string "_ifIndex_" and the PortIfIndex, followed by the string "_RemIp_" and the RemoteAddress. The Python *prepid* function is applied to ensure uniqueness. An example would be *Port_13_ifIndex_2_RemIp_00_0C_41_9D_D3_81* .

```
# prepId function ensures that results are all unique - will add _1, _2 etc
to achieve this
            om.id = self.prepId("Port_" + str(om.Port) + "_ifIndex_" +
str(om.PortIfIndex) + "_RemIp_" + str(om.RemoteAddress))
```

Having cycled around these attribute mappings for the data for the first port, the object map is appended to the relationship map, and the next set of port data is processed.

```
                rm.append(om)
        return rm
```

The second modeler plugin for the ZenPack is trivial in comparison but demonstrates a useful feature and a neat trick. The BridgeDeviceMib.py plugin will be activated for switch devices but will deliver device-wide information, rather than port component information.

The BRIDGE MIB delivers:

- .1.3.6.1.2.1.17.1.1.0 dot1dBaseBridgeAddress
- .1.3.6.1.2.1.17.1.2.0 dot1dBaseNumPorts

Now consider the standard information that is displayed for any Zenoss device on its Status page. This includes a number of standard device properties such as:

- Tag number
- Serial number

- Rack Slot

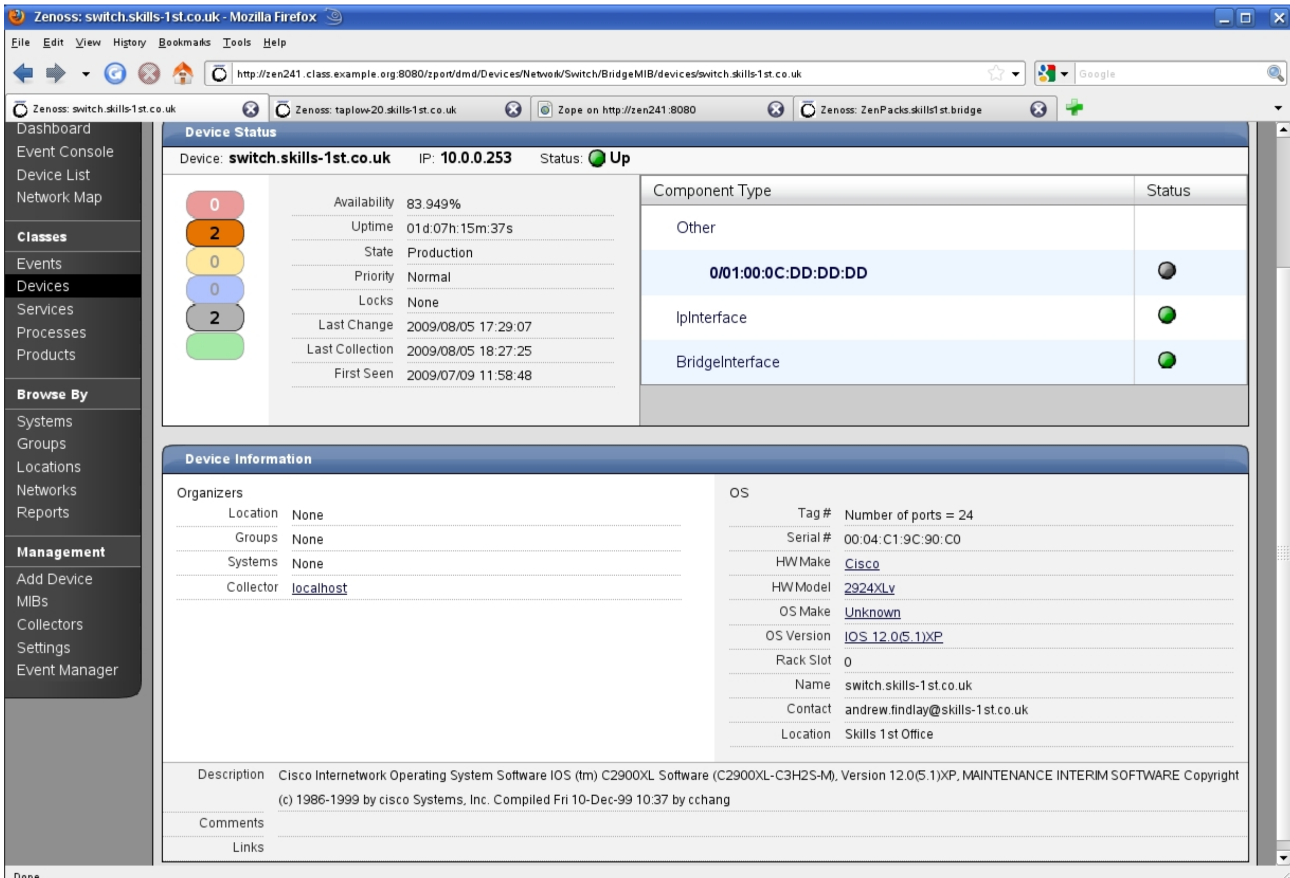

Figure 41: Standard Status page for any Zenoss device

The Tag number is not used normally for switch devices neither is the Serial number field populated; however, the Status page for a device automatically displays data for these fields, if values exist. This is the trick that the BridgeDeviceMib plugin will use. These fields will have data mapped from the dot1dBaseBridgeAddress and dot1dBaseNumPorts OIDs described above.

So, how to get the OIDs into the relevant standard device attributes? Zenoss provides a number of *setter methods* for standard attributes, including *setHWSerialNumber* and *setHWTag* (see the Zenoss Wiki – Diving into the Device Model at http://community.zenoss.org/trac-zenpacks/wiki/DeviceModel for more information on both device setters and properties) . The really useful feature that this plugin demonstrates is that SNMP data can not only be mapped to object attributes; it can also be mapped to setter methods.

jane@zen241:~ - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
##########################################################################
#
# BridgeDeviceMib modeler plugin
#
##########################################################################

__doc__="""BridgeDeviceMib

BridgeDeviceMib gets number of ports and base MAC address for switch supporting Bridge MIB

$Id: $"""

__version__ = '$Revision: $'[11:-2]

from Products.DataCollector.plugins.CollectorPlugin import SnmpPlugin, GetTableMap, GetMap
from Products.DataCollector.plugins.DataMaps import ObjectMap

class BridgeDeviceMib(SnmpPlugin):

#    relname = "BridgeInt"
    modname = "ZenPacks.skills1st.bridge.BridgeDevice"
#    compname = "BridgeDevice"

# snmpGetMap gets scalar SNMP MIBs (single values)
#  Use .1.3.6.1.2.1.17.1.1 ( dot1dBaseBridgeAddress) to populate the Serial No
#  and 1.3.6.1.2.1.17.1.2 ( dot1dBaseNumPorts ) to populate the Hardware tag
#  setHWSerialNumber and setHWTag are standard methods on any Device

    snmpGetMap = GetMap({
        '.1.3.6.1.2.1.17.1.1.0' :  'setHWSerialNumber',
        '.1.3.6.1.2.1.17.1.2.0' :  'setHWTag',
        })


    def process(self, device, results, log):
        """collect snmp information from this device"""
        log.info('processing %s for device %s', self.name(), device.id)
        #Collect Physical Port Forwarding Table
        getdata, tabledata = results

# Uncomment next 2 lines for debugging when modeling
        log.warn( "Get Data= %s", getdata )
        log.warn( "Table Data= %s", tabledata )
        om = self.objectMap(getdata)
        om.setHWSerialNumber = self.asmac(om.setHWSerialNumber)
        om.setHWTag = "Number of ports = " + str(om.setHWTag)
        return om

"BridgeDeviceMib.py" 48L, 1669C written                                 3,17
```

Figure 42: BridgeDeviceMib.py modeler plugin to gather device-wide information for a switch

This modeler populates data into the device specified by ZenPacks.skills1st.bridge.BridgeDevice ie. the device itself, not a component of the device.

Scalar data is gathered using snmpGetMap (whereas the BridgeInterfaceMib plugin used snmpGet**Table**Maps). Note that the OIDs need the trailing *.0* on the end.

```
# snmpGetMap gets scalar SNMP MIBs (single values)
#  Use .1.3.6.1.2.1.17.1.1 ( dot1dBaseBridgeAddress) to populate Serial No
```

```
    #  and 1.3.6.1.2.1.17.1.2 ( dot1dBaseNumPorts ) to populate Hardware tag
    #  setHWSerialNumber and setHWTag are standard methods on any Device

        snmpGetMap = GetMap({
            '.1.3.6.1.2.1.17.1.1.0' :  'setHWSerialNumber',
            '.1.3.6.1.2.1.17.1.2.0' :  'setHWTag',
            })
```

The OID values are mapped to the setHWSerialNumber and setHWTag setter methods, respectively.

In this plugin the tabledata in

```
            getdata, tabledata = results
```

will be empty (it is perfectly possible to have modeler plugins that get both scalar data and table data in the same modeler).

A single object mapping takes place, rather than a looped relationship mapping, and the data is processed slightly for readability.

```
            om = self.objectMap(getdata)
            om.setHWSerialNumber = self.asmac(om.setHWSerialNumber)
            om.setHWTag = "Number of ports = " + str(om.setHWTag)
            return om
```

The result is demonstrated in Figure 41.

4.3.7 Creating the skins files

Having created new object classes for different device types and modeler plugins to collect configuration data for those devices, we now need to design web pages to display the data.

Since the new objects are a device object (BridgeDevice) and a contained component object (BridgeInterface), a new tab is required to augment the standard device tabs. This new "Bridge Interfaces" tab will have details of the individual ports; further, clicking on an individual port will result in a web page showing performance data for that port. So, three new elements are required.

New **tabs** are created in the object class files; the **contents** of those pages are in skins files. Thus BridgeDevice.py copied all standard device tabs and added an extra tab whose label is *Bridge Interfaces* and whose skins file is called *BridgeDeviceDetail* (note that there is no .pt here but the actual file under the skins directory hierarchy must end in .pt).

```
jane@zen241:~ - Shell - Konsole <3>
Session  Edit  View  Bookmarks  Settings  Help
######################################################################
#
# BridgeDevice object class
#
######################################################################

from Globals import InitializeClass
from Products.ZenRelations.RelSchema import *
from Products.ZenModel.Device import Device
from Products.ZenModel.ZenossSecurity import ZEN_VIEW
from copy import deepcopy


class BridgeDevice(Device):
    "A Bridge Device"

    _relations = Device._relations + (
        ('BridgeInt', ToManyCont(ToOne,
            'ZenPacks.skills1st.bridge.BridgeInterface', 'BridgeDev')),
        )

    factory_type_information = deepcopy(Device.factory_type_information)
    factory_type_information[0]['actions'] += (
            { 'id'              : 'BridgeInt'
            , 'name'            : 'Bridge Interfaces'
            , 'action'          : 'BridgeDeviceDetail'
            , 'permissions'     : (ZEN_VIEW, ) },
            )


    def __init__(self, *args, **kw):
        Device.__init__(self, *args, **kw)
        self.buildRelations()


InitializeClass(BridgeDevice)
~
"BridgeDevice.py" 36L, 1080C written                          3,27          All
Shell
```
Figure 43: BridgeDevice.py object class file - note the action called BridgeDeviceDetail

Similarly, BridgeInterface.py defines three tabs, one of which is specific to the ZenPack (the viewBridgeInterface action) and two standard tabs from $ZENHOME/Products/ZenModel/skins/zenmodel (objTemplates and viewHistory). Note that BridgeInterface.py does not copy any existing tabs and that the *product* parameter must indicate the last part of the ZenPack name (*bridge* in this case). The *immediate_view* parameter can be used to define which tab is initially opened.

jane@zen241:~ - Shell - Konsole <3>

Session Edit View Bookmarks Settings Help

```
#################

_relations = (
    ("BridgeDev", ToOne(ToManyCont,
        "ZenPacks.skills1st.bridge.BridgeDevice", "BridgeInt")),
    )

factory_type_information = (
    {
        'id'             : 'BridgeInterface',
        'meta_type'      : 'BridgeInterface',
        'description'    : """Bridge Interface info""",
        'product'        : 'bridge',
        'immediate_view' : 'viewBridgeInterface',
        'actions'        :
        (
            { 'id'            : 'status'
            , 'name'          : 'Bridge Interface Status'
            , 'action'        : 'viewBridgeInterface'
            , 'permissions'   : (ZEN_VIEW, )
            },
            { 'id'            : 'perfConf'
            , 'name'          : 'Bridge Interface Template'
            , 'action'        : 'objTemplates'
            , 'permissions'   : (ZEN_CHANGE_SETTINGS, )
            },
            { 'id'            : 'viewHistory'
            , 'name'          : 'Modifications'
            , 'action'        : 'viewHistory'
            , 'permissions'   : (ZEN_VIEW, )
            },
        )
      },
    )
```

Figure 44: BridgeInterface.py object class file showing tab definitions

Skins files defining web pages live under the skins/ZenPacks.skills1st.bridge subdirectory (for this ZenPack). As a general guideline, start creating skins files by looking for a sample file (on the forum, the wiki, or in the standard $ZENHOME/Products/ZenModel/skins/zenmodel directory); copy the sample and modify it to suit. Consult Chapter 13 of the Zenoss Developer's Guide 2.4 for lots of explanations about skins files.

If you are not familiar with the different techniques of TAL, METAL, TALES, HTML and ZPT , it can be very confusing as to what is going on!

- HyperText Markup Language (HTML) - is the most basic formatting language available on the Web, and some version of HTML is understood by every Web browser. HTML is in practice a sloppy variant of eXtensible Markup Language (XML) which divides up a page into elements (tags such as title, head or h3) and content (for example, the things that you actually care about). Common HTML tags found in Zenoss skins files include:
    - <th> table header
    - <td> table data

- <tr> table row
-
 break
- <block> creates larger structures that can include other blocks
- <form> for user input
- <input> input directive

- Zope Page Templates (ZPT) - are in essence HTML pages which are well-formed and have extra XML attributes (ie the bits after the element name in-between the < and > characters). The extra XML bits (attributes) are not a part of any HTML standard and are ignored by HTML editors, meaning that ZPT pages live happily with HTML. These attributes and the programming functionality that they deliver are called the Template Attribute Language (TAL). Zenoss skins files all have a *.pt* extension for Page Template.
- Template Attribute Language (TAL) - the TAL attributes allow you to add dynamic content using information from inside the Zope database (ZODB). From a Zenoss perspective, this allows you to write a query that you can use to build a table, or show different items depending on what objects or devices exist in a particular state. In other words, TAL is the Zope way of accomplishing what you would normally need to do in a CGI inside of a plain web server like Apache. It should be noted that inside TAL it is also possible to use a restricted subset of Python. The restrictions include not being able to load certain standard libraries, as well as operations like reading and writing to disk. This is done intentionally for security reasons. See http://docs.zope.org/zope2/zope2book/source/AppendixC.html for a Zope Page Template reference. TAL includes statements such as:
  - tal:define define variables
  - tal:condition test conditions
  - tal:content replace the content of an element
  - tal:repeat repeat an element
  - tal:replace replace content of an element
  - tal:attributes dynamically change element attributes
- Macro Expansion for TAL (METAL) - because TAL is hidden away inside HTML, there's no way to reuse blocks of HTML and TAL for your site just by using TAL. METAL allows page templates to define *macros* (which are essentially sub-templates that may be called by other templates) and *slots* (which may be filled by other templates). Several METAL macros are provided with Zenoss such as:
  - page1 provides web page with breadcrumbs and content
  - page2 page 1 plus standard breadcrumbs and navigation tabs

- page3 page 1 plus standard breadcrumbs, no tabs
- zentable creates tables of data for display
- navbodypagedevice macro to support sorting, filtering, multi-pages

- TAL Expression Syntax (TALES) - TALES allows access to the template's namespace, including useful properties such as the *here* context object. TALES accepts paths (e.g. *here/id*) which it resolves into object properties. It will attempt to resolve the final path element as a key index, a key name, an attribute, or a method. For example, if getSomething() is a method on the context, here/getSomething will return the result of that method. TALES statements are what normally provides the dynamic content for a page template, delivering data from the ZODB database.

This ZenPack has referred to two ZenPack-specific skins files; BridgeDeviceDetail.pt from BridgeDevice.py and viewBridgeInterface.pt from BridgeInterface.py.

viewBridgeInterface.pt is simple and in fact, only uses a standard METAL macro to display any performance graphs that have been customised for a port interface.

```
<tal:block metal:use-macro="here/templates/macros/page2">
<tal:block metal:fill-slot="contentPane">

<form method=post
    tal:define="manager here/isManager" >
    <input type="hidden" name="zenScreenName"
        tal:attributes="value template/id" />

</form>
<tal:block tal:condition="here/monitored" >
<table metal:use-macro="here/viewPerformanceDetail/macros/objectperf" />
</tal:block>
</tal:block>
</tal:block>

"viewBridgeInterface.pt" 16 lines --6%--                    1,1           All
```

Figure 45: viewBridgeInterface.pt skins file to display performance graphs for a port interface

The first line calls a METAL macro to define a page with standard breadcrumbs and automatic tabs (*here/templates/macros/page2*).

There is a TAL condition to check that the device is being monitored.

The standard macro to display performance data for an object is called (*here/viewPerformanceDetail/macros/objectperf*). This macro can be found in $ZENHOME/Products/ZenModel/skins/zenmodel/viewPerformanceDetail.pt .

The resulting page is shown in Figure 46. At this stage, do not worry about the contents of the graph, simply that the graph is displayed with data. Performance data will be looked at in more detail later.

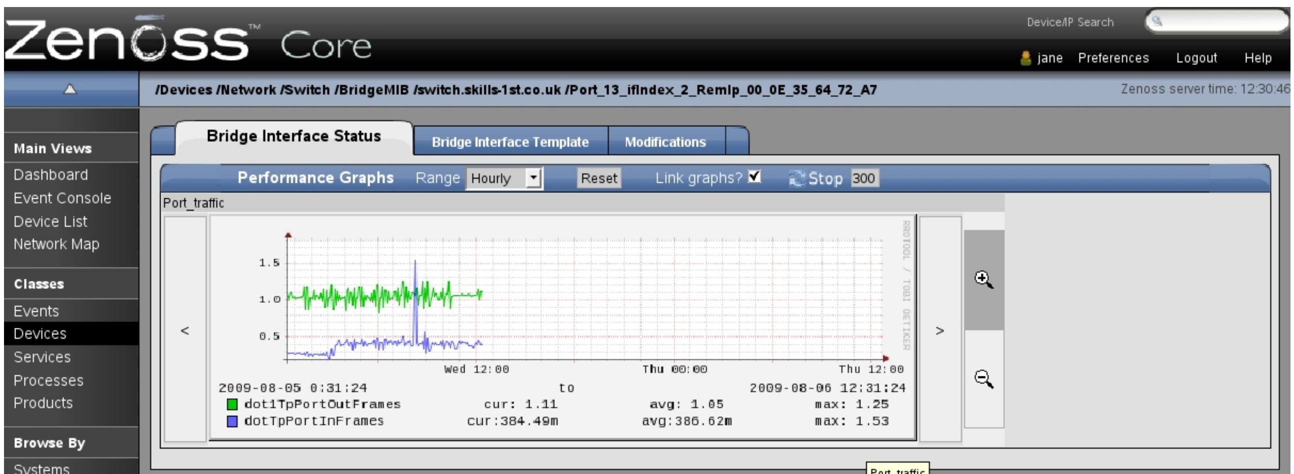

Figure 46: Performance graph for a bridge interface

The second skins file is more complex. It is best described alongside a screenshot of the result – see Figure 47.

Port Name	Port Number	Port Interface Index	Remote Address	Remote IP Address	Remote Hostname	Remote Interface Description	Port Status Value	Port Status
13/00:0C:41:9D:D3:81	13	2	00:0C:41:9D:D3:81	[]	[]	[]	Learned (3)	
13/00:0E:35:64:72:A7	13	2	00:0E:35:64:72:A7	[]	[]	[]	Learned (3)	
13/00:11:25:80:1C:4F	13	2	00:11:25:80:1C:4F	['10.0.0.121']	['bino.skills-1st.co.uk']	['eth1']	Learned (3)	
40/00:04:C1:9C:90:C0	40	-1	00:04:C1:9C:90:C0	['10.0.0.253']	['switch.skills-1st.co.uk']	['VLAN1']	Not active (4)	
40/00:04:C1:9C:90:C1	40	-1	00:04:C1:9C:90:C1	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_1']	Not active (4)	
40/00:04:C1:9C:90:C2	40	-1	00:04:C1:9C:90:C2	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_2']	Not active (4)	
40/00:04:C1:9C:90:C3	40	-1	00:04:C1:9C:90:C3	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_3']	Not active (4)	
40/00:04:C1:9C:90:C4	40	-1	00:04:C1:9C:90:C4	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_4']	Not active (4)	
40/00:04:C1:9C:90:C5	40	-1	00:04:C1:9C:90:C5	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_5']	Not active (4)	
40/00:04:C1:9C:90:C6	40	-1	00:04:C1:9C:90:C6	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_6']	Not active (4)	
40/00:04:C1:9C:90:C7	40	-1	00:04:C1:9C:90:C7	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_7']	Not active (4)	
40/00:04:C1:9C:90:C8	40	-1	00:04:C1:9C:90:C8	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_8']	Not active (4)	
40/00:04:C1:9C:90:C9	40	-1	00:04:C1:9C:90:C9	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_9']	Not active (4)	
40/00:04:C1:9C:90:CA	40	-1	00:04:C1:9C:90:CA	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_10']	Not active (4)	
40/00:04:C1:9C:90:CB	40	-1	00:04:C1:9C:90:CB	['10.0.0.253']	['switch.skills-1st.co.uk']	['FastEthernet0_11']	Not active (4)	

Figure 47: Web page produced by BridgeDeviceDetail.pt

The objective is to produce a single table with information for each port on a separate line. The readability of adjacent lines is enhanced by alternating the background colour.

Some of the port information is simply object attribute values, such as Port, PortIfIndex and RemoteAddress - the unique attributes defined in BridgeInterface.py.

Some of the information is constructed using methods of the object, again defined in BridgeInterface.py; these include Port Name, Remote IP Address, Remote Hostname and Remote Interface Description.

The last two fields of the table present the value of the object attribute PortStatus in two different ways. Fundamentally, if the status is 3 (learned) then it is deemed to be “active”. The last field is a green bullet for an active port; otherwise the bullet is red. The previous field presents the PortStatus value but rather than just presenting the numeric value (3, 4, 5, etc), it also provides a decode for the number (*Learned* or *Not Active*).

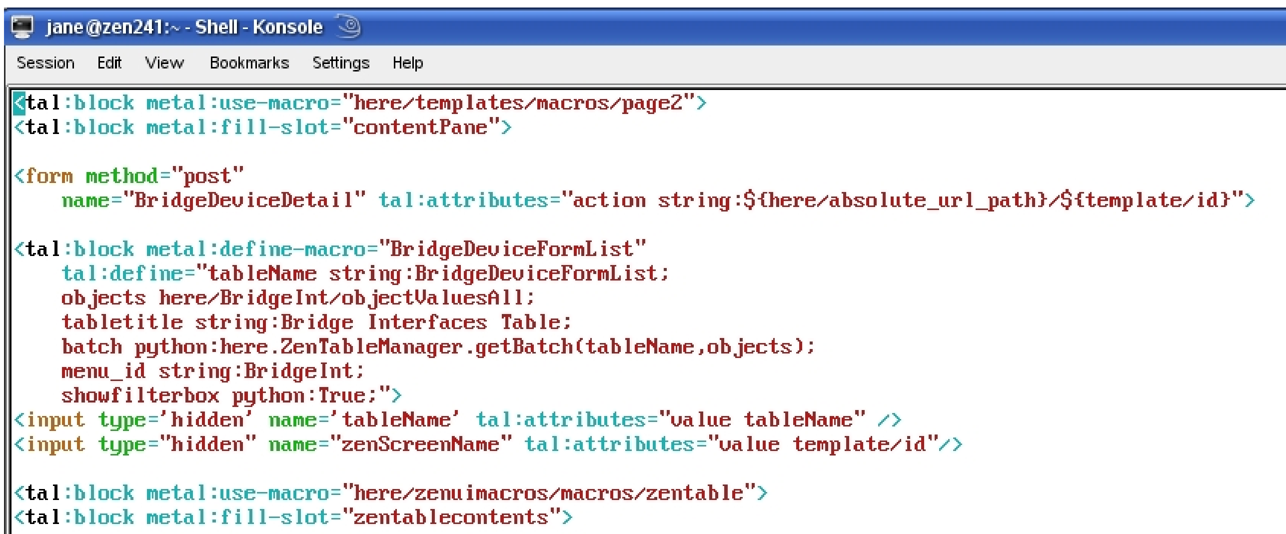

Figure 48: BridgeDeviceDetail.pt (part 1) showing page type and BridgeDeviceFormList macro

The first line of BridgeDeviceDetail.pt uses the page2 macro again for a page with breadcrumbs and tabs.

The *BridgeDeviceFormList* macro is defined to get all the objects from the device's BridgeInt relationship (*here/BridgeInt/objectValuesAll*) and supply them in a table. Since there may be many interfaces, the filter box (at the top right of the GUI page) should be enabled (*showfilterbox python:True*).

```
<tal:block metal:define-macro="BridgeDeviceFormList"
    tal:define="tableName string:BridgeDeviceFormList;
    objects here/BridgeInt/objectValuesAll;
    tabletitle string:Bridge Interfaces Table;
    batch python:here.ZenTableManager.getBatch(tableName,objects);
    menu_id string:BridgeInt;
    showfilterbox python:True;">
```

The second part of the skins file defines the table headers with their layout.

Figure 49: BridgeDeviceDetail.pt (part 2) showing table headers layout

The line:

```
<tr tal:condition="objects">
```

starts the definition of the table row (<tr matched by the closing </tr> 5 lines from the end of Figure 49), and uses a TAL statement to ensure that the variable *objects* was actually populated from *here/BridgeInt/objectValuesAll* in the earlier section. If *objects* is null then the remainder of the <tr> row definition will be ignored..

There are lots of permutations for structuring header and data lines of a table. The comments in Figure 49 explain some of the consequences. A flexible way is to use lines like the following:

```
<th tal:replace="structure
python:here.ZenTableManager.getTableHeader(tableName,'Port','Port Number',
attributes='width=15')"/>
```

The table header (< th ... />) uses a TAL replace statement to use Python to access the table defined with the object attribute values and to use the string 'Port Number' as the column header for Port data, allowing 15 characters width.

Another alternative would be to use a table data (<td ... />) HTML tag but this seems to result in a table where columns are not sortable:

```
<td class="tableheader" align=left>Port Name</td>
```

If the attributes are more complex or extensive, they can be declared separately:

```
<th tal:define="attributes string:'width=20'"
   tal:replace="structure
python:here.ZenTableManager.getTableHeader(tableName,'RemoteAddress',
'Remote Address', attributes=attributes)"/>
```

If the *objects* variable is null then a warning message is displayed:

```
        <tr tal:condition="not:objects">
            <th class="tableheader" align="left" colspan="9">
                No Interfaces found. Double check you have the correct
    collector plugin and you have remodeled.
            </th>
        </tr>
```

Next, a block is set up that will repeatedly output one row of the table for each port, with alternate lines having a different background.

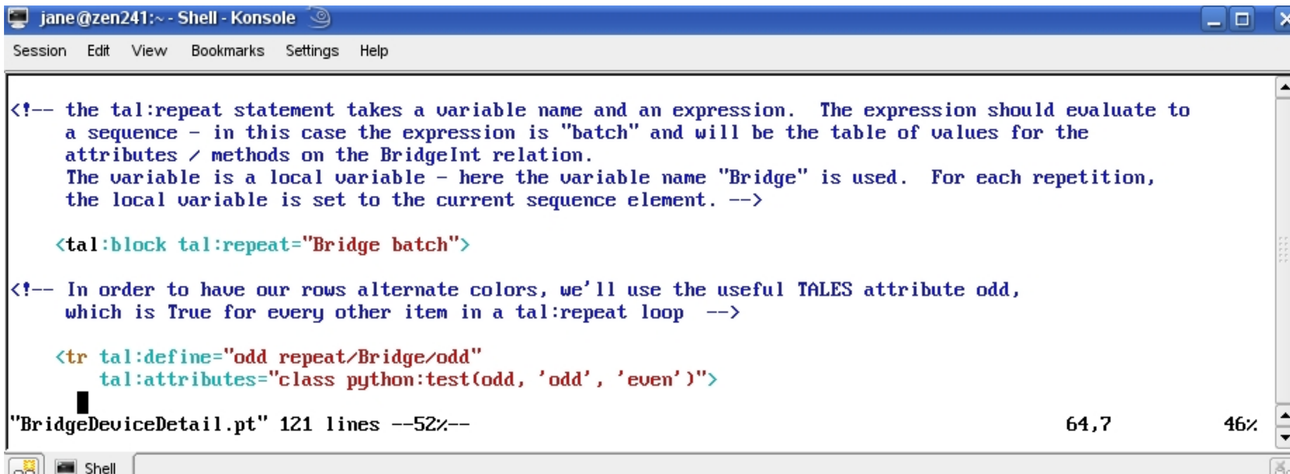

Figure 50: BridgeDeviceDetail.pt (part 3) showing the controls for the data rows of the port table

The *tal:repeat* statement takes a variable name and an expression. The expression should evaluate to a sequence - in this case the expression is *batch* (defined earlier in the BridgeDeviceFormList macro) and will be the table of values for the attributes / methods on the BridgeInt relationship. The variable is a local variable - here the variable name *Bridge* is used. For each repetition, the local variable is set to the current sequence element.

```
    <tal:block tal:repeat="Bridge batch">
```

A standard TALES attribute, *odd*, can be used which evaluates to True for every other item in a tal:repeat loop. It provides different background colours for alternate lines. This code fraction also shows the start of the table row definition (<tr).

```
    <tr tal:define="odd repeat/Bridge/odd"
            tal:attributes="class python:test(odd, 'odd', 'even')">
```

The next section provides the data values for a row of the table.

jane@zen241:~ - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
    <td class="tablevalues">
      <a class=tablevalues tal:content="Bridge/name"
      tal:attributes="href Bridge/getPrimaryUrlPath">BlaBla</a>
    </td>

    <td class="tablevalues">
      <a class=tablevalues tal:content="Bridge/Port"
      tal:attributes="href Bridge/getPrimaryUrlPath">BlaBla</a>
    </td>
    <td class="tablevalues">
      <a class=tablevalues tal:content="Bridge/PortIfIndex"
      tal:attributes="href Bridge/getPrimaryUrlPath">BlaBla</a>
    </td>
    <td class="tablevalues">
      <a class=tablevalues tal:content="Bridge/RemoteAddress"
      tal:attributes="href Bridge/getPrimaryUrlPath">BlaBla</a>
    </td>
    <td class="tablevalues" tal:content="Bridge/getIpRemoteAddress">Nomatch </td>
    <td class="tablevalues" tal:content="Bridge/getIpRemoteHostname">Nomatch </td>
    <td class="tablevalues" tal:content="Bridge/getIpRemoteIfDesc">Nomatch </td>
    <td class="tablevalues"
            tal:content="python:Bridge.PortStatus==3 and 'Learned (3)' or 'Not active (' + str(Bridge.PortSt
atus) + ')'">
                Learned
    </td>

    <td class="tablevalues" align="center">
    <img border="0"
      tal:attributes="src python:test(Bridge.PortStatus==3,
          here.getStatusImgSrc(0),
          here.getStatusImgSrc(3))" />
    </td>

   </tr>
  </tal:block>
"BridgeDeviceDetail.pt" 121 lines --53%--                                        65,6          74%
```

Shell

Figure 51: BridgeDeviceDetail.pt (part 4) showing the data values for the port table

Each table data (<td>) tag uses a tal:content statement to reference either an attribute or a method on the BridgeInterface object to deliver a data value. Remember that *Bridge* is the local variable that takes the next set of values from the port table, each time round the tal:repeat loop. Extra attributes can be specified, if required.

```
<td class="tablevalues">
        <a class=tablevalues tal:content="Bridge/name"
        tal:attributes="href Bridge/getPrimaryUrlPath">BlaBla</a>
      </td>
```

The first PortStatus data value, rather than simply showing the numeric value from the object, will also “translate” the numeric value into a more useful human representation. This uses a Python test:

```
<td class="tablevalues"
           tal:content="python:Bridge.PortStatus==3 and 'Learned (3)' or
            'Not active (' + str(Bridge.PortStatus) + ')'">
           Learned
</td>
```

If the PortStatus of this switch port is 3 then the output will be the string 'Learned (3)'; otherwise the output will be the string 'Not active' concatenated with the string representation of the value of PortStatus, concatenated with a closing ')' .

The final data column in the table of data is a red or green bullet representing either an active port (with PortStatus = 3) or a non-active port.

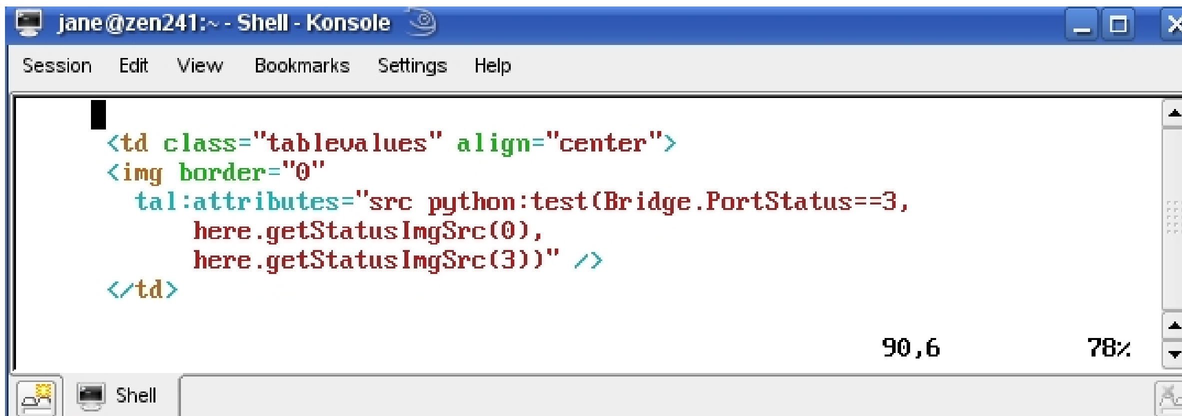

Figure 52: BridgeDeviceDetail.pt (part 5) showing code to produce coloured bullets to represent PortStatus

This is code used in several places in the standard Zenoss skins files. Again, it uses a Python test to evaluate PortStatus and then uses here.getStatusImgSrc(0) to represent a green bullet and here.getStatusImgSrc(3) for a red bullet.

The remainder of BridgeDeviceDetail.pt has the closing table row tag and the closing block tag for the data rows. The standard METAL macro *navbodypagedevice* is called to ensure that the table can be searched, the columns can be ordered and large numbers of rows will correctly be split into pages. Note that the earlier *navtool* macro does not seem to implement filtering and paging correctly. The last few lines of the file are the closing tags for blocks and the overall form.

```
jane@zen241:~ - Shell - Konsole
Session  Edit  View  Bookmarks  Settings  Help

     </tr>
    </tal:block>
    <tr>
        <td colspan="9" class="tableheader" align='center'>

<!-- The here/zenTableNavigation/macros/navtool doesn't seem to support table filtering and "Show all"
     although it does support sorting.  The here/zenTableNavigation/macros/navbodypagedevice
     seems to support sorting, filtering and breaking into pages / show all    -->

<!--            <form metal:use-macro="here/zenTableNavigation/macros/navtool"></form> -->
            <span metal:use-macro="here/zenTableNavigation/macros/navbodypagedevice" />
        </td>
    </tr>

<!-- END TABLE CONTENTS -->

</tal:block>
</tal:block>
</tal:block>

</form>

</tal:block>
</tal:block>
"BridgeDeviceDetail.pt" 121 lines --100%--                              121,6          Bot
Shell
```

Figure 53: *BridgeDeviceDetail.pt (part 6) with closing tags and the navbodypagedevice macro call*

4.3.8 Linking development mode elements with source mode elements

At this stage we have:

- A new device class, *BridgeMIB*, a subclass of /Devices/Network/Switch, created via the GUI and added to the ZenPack in Development mode
- Some MIBs added to the ZenPack in Development mode
- Two new object class files, *BridgeDevice.py* and *BridgeInterface.py* in the base directory of the ZenPack, created in source mode
- Two modeler plugins, *BridgeInterfaceMib.py* and *BridgeDevice.py* in the modeler/plugins subdirectory of the base ZenPack directory, created in source mode
- Two skins files, *BridgeDeviceDetail.pt* and *viewBridgeInterface.pt* in the skins/ZenPacks.skills-1st.bridge subdirectory of the base ZenPack directory, created in source mode

Nothing yet links the new device class with the object classes and their associated modelers. This is achieved using the GUI.

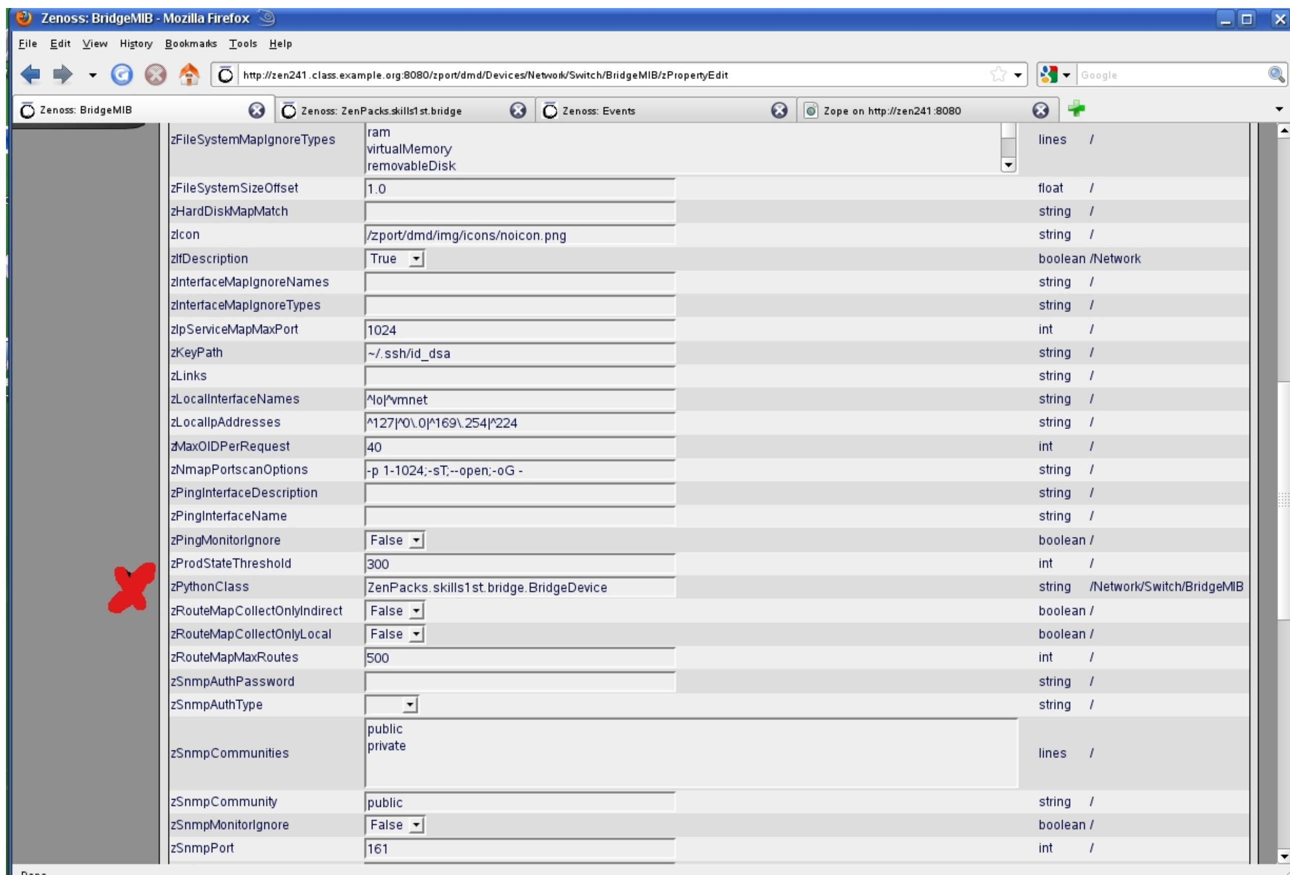

Figure 54: Linking a device class with the object class file that describes its unique properties

To associate the device class, *BridgeMIB*, with the object class *BridgeDevice*, simply modify the zProperty **zPythonClass,** either for an individual device or for a subclass of devices. Remember to save the modifications. The zPythonClass should be the fully-qualified *object* name. In this case, the object is defined in this ZenPack so the zPythonClass is *ZenPacks.skills1st.bridge.BridgeDevice* (no .py on the end). Don't forget to save the modification. There is no direct association here with the BridgeInterface class as that is a contained object class of BridgeDevice.

The second link required, is between the device class and the modeler plugins to be deployed for that class. This is done from the *More -> Collector Plugins* menu of either an individual device or a device class. If the plugin source code is valid then the name of the plugin should automatically appear in the *Add Fields* list. Required plugins are dragged to the top, selected area and can be reordered simply by dragging them around. Again, don't forget the *Save* button.

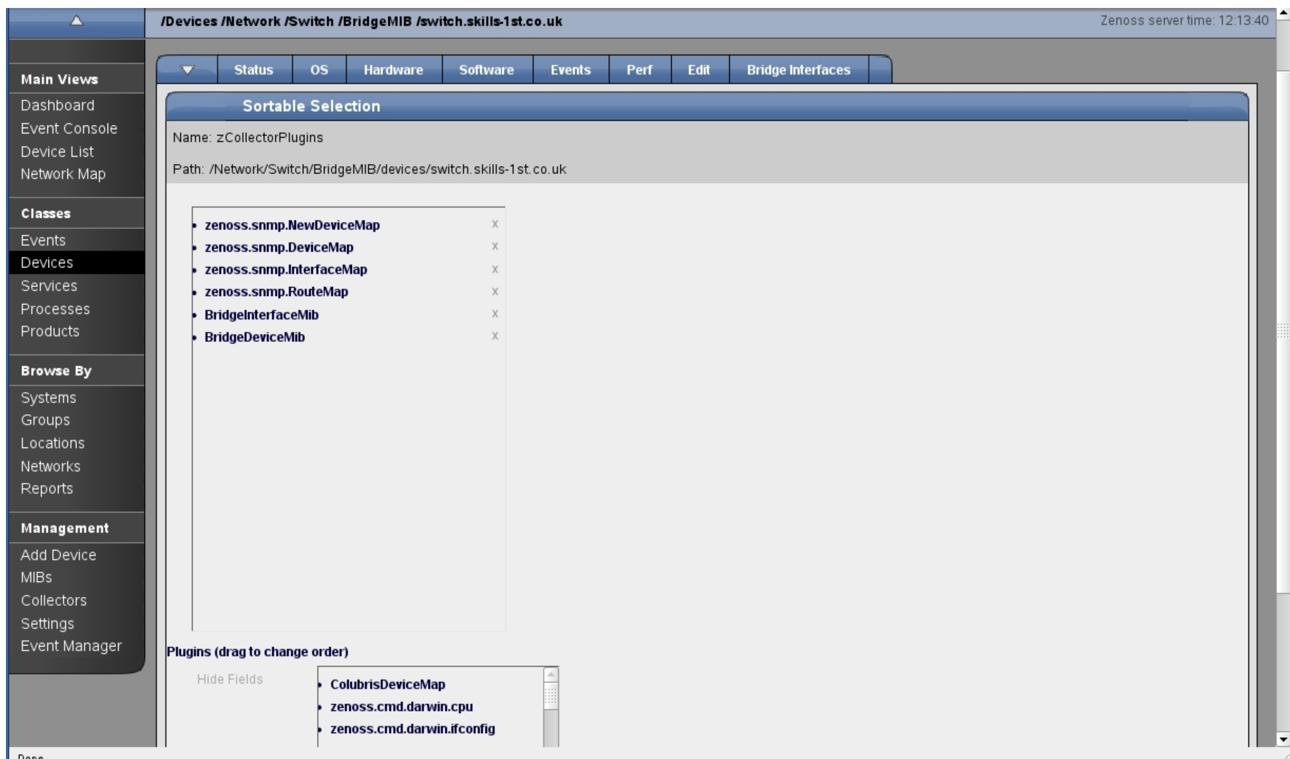

Figure 55: Associating a device with a set of modeler plugins

To propagate these associations to the ZenPack, return to the ZenPacks tab under the *Settings* menu, select the ZenPack, and use the table drop-down menu to *Export ZenPack*. In addition to creating the egg file (*ZenPacks.skills1st.bridge-1.0-py2.4.egg*) for the ZenPack in *$ZENHOME/export*, exporting also updates the *object.xml* file in the *objects* subdirectory of the ZenPack.

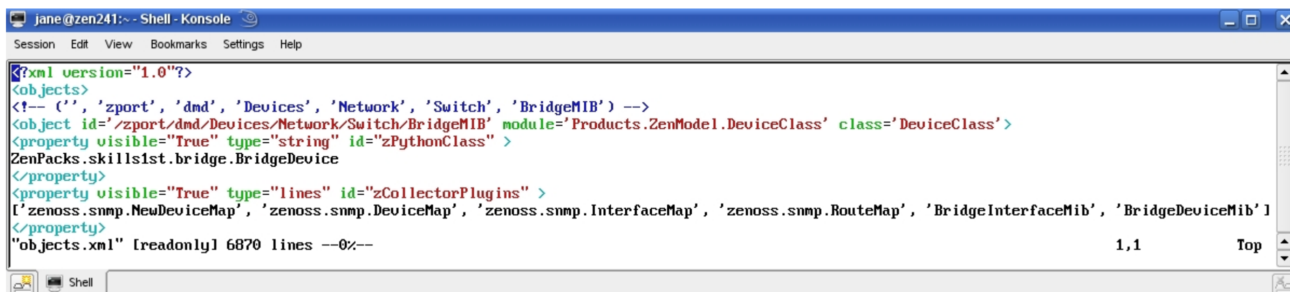

Figure 56: Start of objects.xml showing zPythonClass and zCollectorPlugins

The egg file could now be moved to a different system and loaded there.

5 Gathering Performance Data

Performance data is gathered, usually by either the *zenperfsnmp* daemon (for SNMP data), or by the *zencommand* daemon (for ssh data). Other performance data collectors may be made available by other ZenPacks.

By default, zenperfsnmp runs every 5 minutes; for ssh-collected data, the performance template allows you to specify a collection interval although zencommand only runs every minute, by default. With Zenoss Core, a single zenperfsnmp daemon is available (although it is possible to deploy others); with Zenoss Enterprise, multiple data collectors can be configured fairly easily. This means in a typical Zenoss Core installation, that there really is only one polling interval configuration to collect SNMP performance data. The default of 5 minutes can be changed easily using the *Collectors -> localhost -> Edit* menu, but there is still only one collector (or *monitor*, as it used to be called).

To specify performance data for collection, Zenoss **templates** need to be created and **bound** to a device or device class. A template defines:

- One or more data sources
- One or more data points
- Threshold values, if required
- Graph definitions, if required

5.1 Performance templates for devices

For the Bridge MIB ZenPack, some data may be required pertinent to the whole device; other data will be per-port. Device-wide data can be gathered in the usual manner and will be displayed under the standard *Perf* tab.

For example, the Bridge MIB provides values out of the Spanning Tree Protocol (Stp) subtree of the MIB which gives:

- dot1dStpTimeSinceTopologyChange (TimeTicks) .1.3.6.1.2.1.17.2.3.0
- dot1dStpTopChanges (Counter32) .1.3.6.1.2.1.17.2.3.0

These values give a measure of the number of centi-seconds since the last Stp topology change and the number of topology changes since the last initialise or reboot of the device. Note that both have *.0* on the end – they are scalar MIB values ie. there is only one value for the whole device. A Zenoss template can be configured to collect and graph these values.

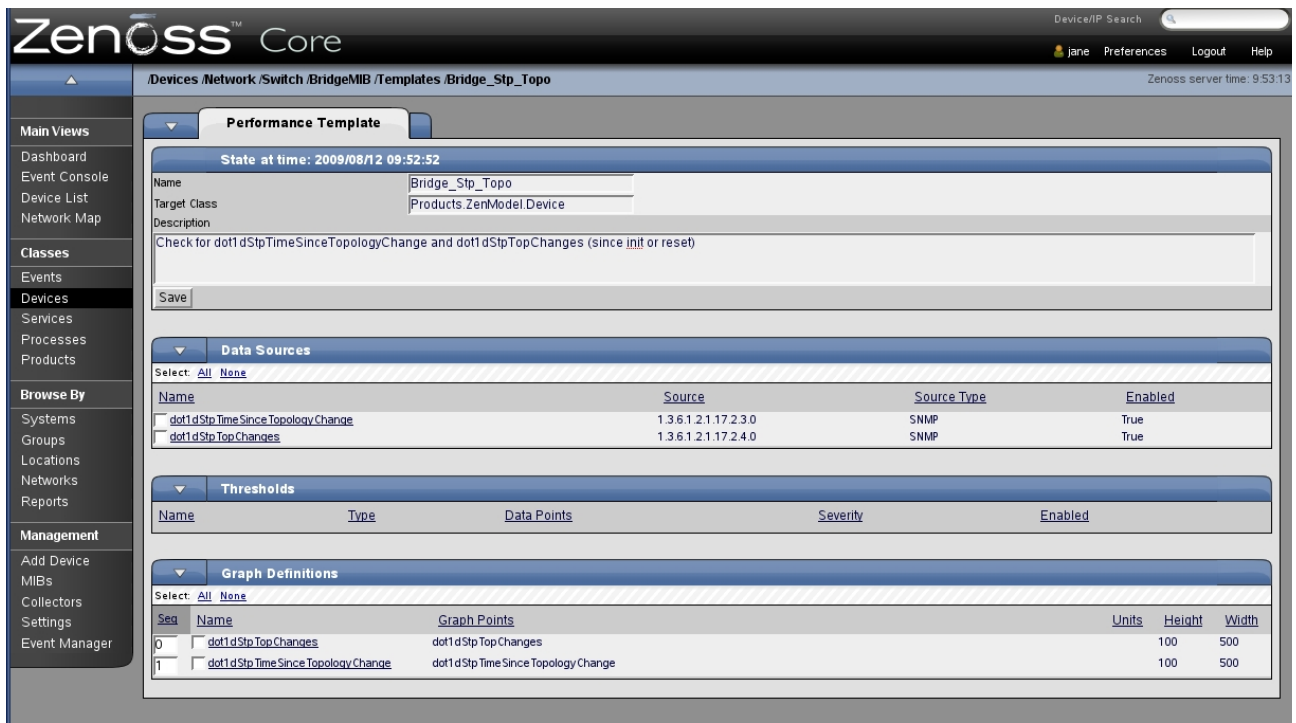

Figure 57: Zenoss performance template to gather Bridge Stp topology change data for the BridgeMIB device class

To activate the template, it must be **bound** either to a device class or to a specific device. Use the Templates tab for a device class or the *More -> Templates* menu for a specific device and then use the table drop-down menu to bind one or more templates.

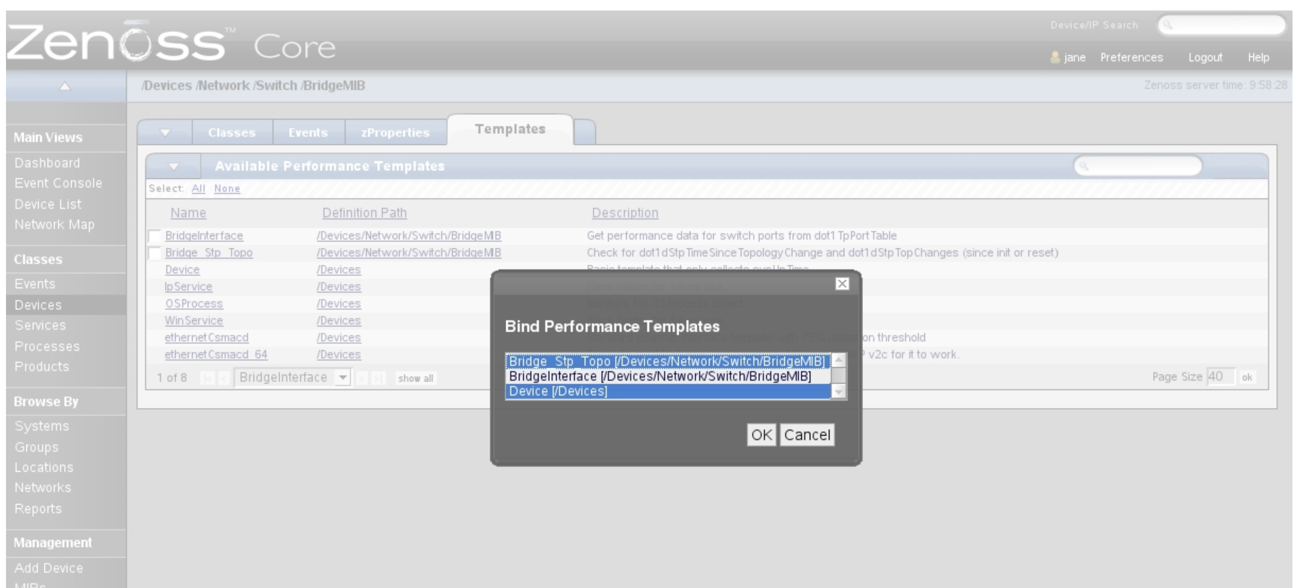

Figure 58: Binding templates to the /Devices/Network/Switch/BridgeMIB device class

You should then see graph outlines under the Perf tab for any device that has this template applied; however it will typically be two zenperfsnmp collection intervals before you start to see data. Note that since dot1dStpTimeSinceTopologyChange is in

units of centi-seconds the graph point has been modified with a Reverse Polish Notation (RPN) expression to convert it to seconds and the *Units* field of the graph definition has been set to *secs*.

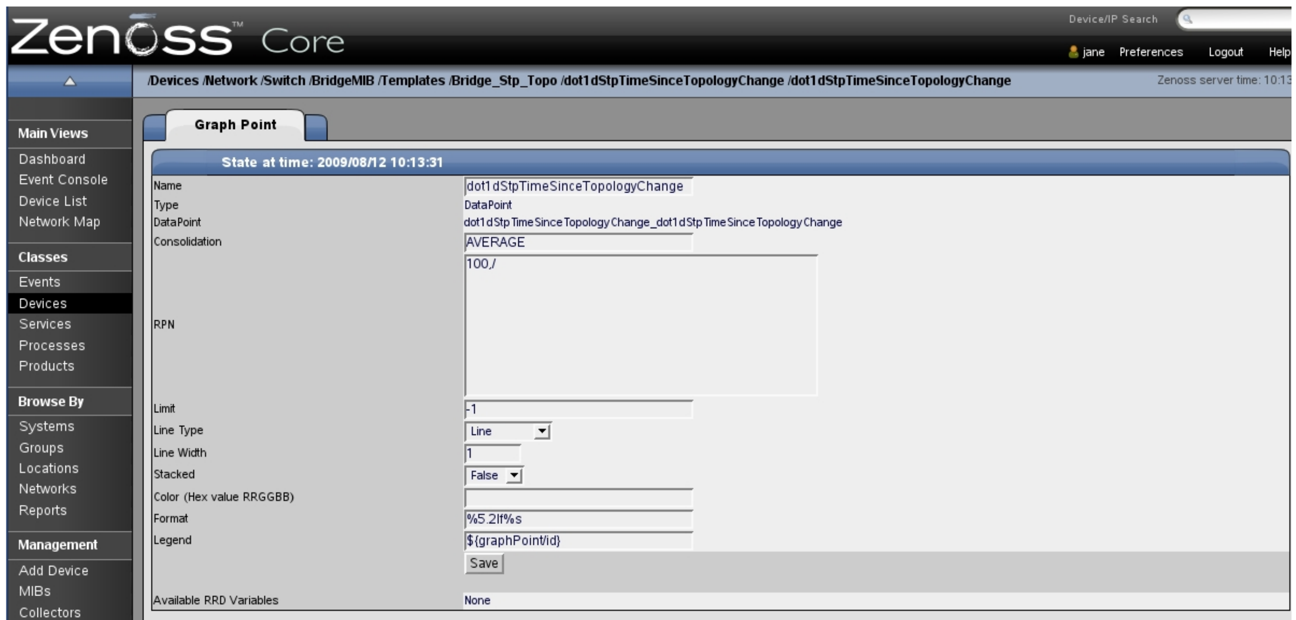

Figure 59: dot1d Stp template showing Reverse Polish Notation (RPN) to change data units

The resultant graph is shown in Figure 60.

Figure 60: Performance graph for switch showing dot1d Stp data

Thus far, none of the new ZenPack functionality has been used though it may be useful to add the Bridge_Stp_Topo template to the ZenPack as shown in Figure 61.

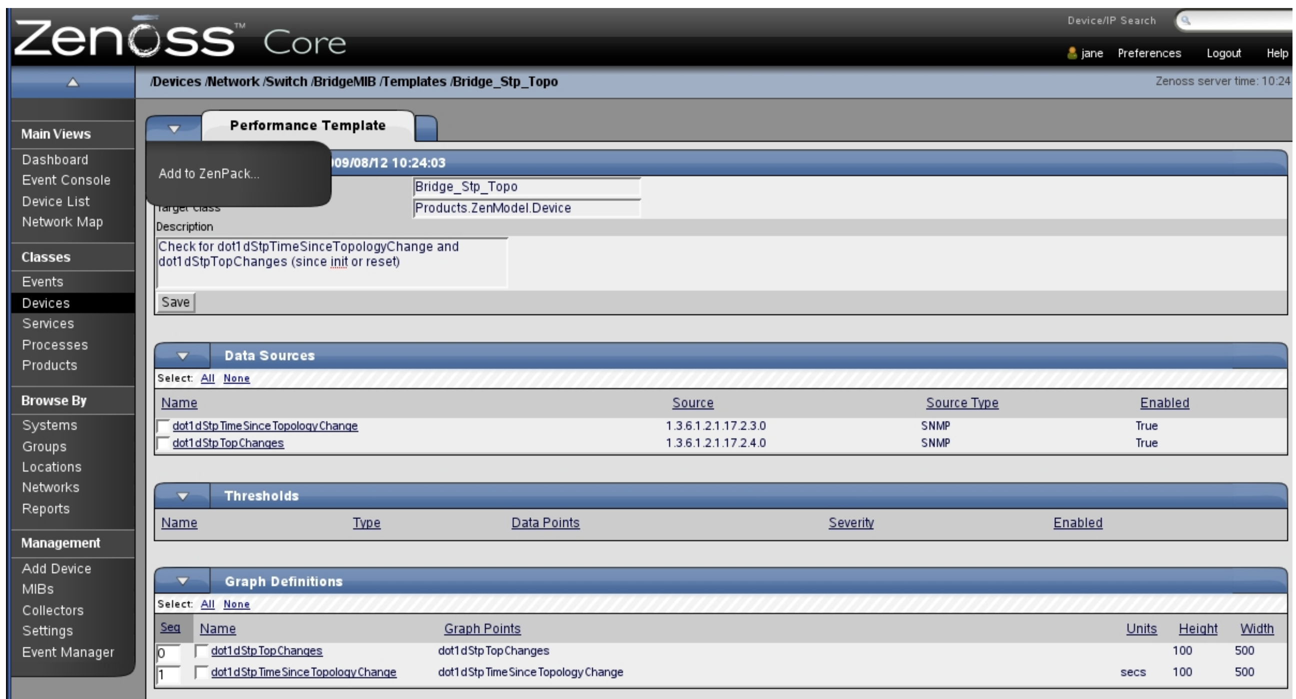

Figure 61: Adding Zenoss performance Template to a ZenPack

Re-exporting the ZenPack will also update the definition of the BridgeMIB device class in *objects/objects.xml*, including the *zDeviceTemplates* zProperty, if you have bound the template to that device class.

5.2 Performance templates for contained devices

To get performance information for the switch **ports** that are supported by the ZenPack, there are two important factors:

- A Zenoss Template **with exactly the same name as a contained, component class object**, will **automatically** be bound to instances of that object. The object class representing a switch port is BridgeInterface; thus a template called BridgeInterface will automatically be bound to such objects.
- When specifying a template for SNMP performance data to be collected, unless the data is a scalar, you do not specify the instance to be collected. The instances are taken from the object class (BridgeInterface) **snmpindex** attribute.

Remember when the BridgeInterfaceMib modeler plugin was created, it populated not only the unique attributes of the BridgeInterface object class, but also populated the inherited attributes of:

- id
- snmpindex

Since most of the useful performance data from the BRIDGE MIB is indexed by the Port value, the snmpindex attribute was set to this value, having first converted the raw data to an integer type. Thus for a Catalyst 2900, the Port values, and hence the snmpindex values, run from 13 to 38.

The Zenoss *zendmd* utility is useful to see the values of objects – see Figure 62 for the code and Figure 63 for an output fragment.

```
>>>
>>>
>>> dev=find('switch.skills-1st.co.uk')
>>> for i in dev.BridgeInt():
...   for key,value in i.__dict__.items():
...     print key,value
...
```

Figure 62: Using zendmd to see values of the attributes of a BridgeInterface object

Note in Figure 62 that you start with a device and then print information for the BridgeInt **relationship** for that device.

```
snmpindex 13
RemoteAddress 00:0C:41:9D:D3:81
id Port_13_ifIndex_2_RemIp_00_0C_41_9D_D3_81
__primary_parent__ <ToManyContRelationship at BridgeInt>
BridgeDev <ToOneRelationship at BridgeDev>
createdTime 2009/07/29 21:02:40.042 GMT+1
_objects ({'meta_type': 'ToOneRelationship', 'id': 'BridgeDev'},)
Port 13
PortStatus 3
snmpindex 13
RemoteAddress 00:11:25:80:1C:4F
id Port_13_ifIndex_2_RemIp_00_11_25_80_1C_4F
__primary_parent__ <ToManyContRelationship at BridgeInt>
BridgeDev <ToOneRelationship at BridgeDev>
createdTime 2009/07/29 21:02:37.981 GMT+1
_objects ({'meta_type': 'ToOneRelationship', 'id': 'BridgeDev'},)
Port 13
PortStatus 3
snmpindex 40
RemoteAddress 00:04:C1:9C:90:C0
id Port_40_ifIndex_-1_RemIp_00_04_C1_9C_90_C0
__primary_parent__ <ToManyContRelationship at BridgeInt>
PortIfIndex -1
createdTime 2009/07/29 21:02:41.107 GMT+1
_objects ({'meta_type': 'ToOneRelationship', 'id': 'BridgeDev'},)
Port 40
BridgeDev <ToOneRelationship at BridgeDev>
PortStatus 4
```

Figure 63: Output of the zendmd code to view attributes of a BridgeInterface object

The BridgeInterfaceMib modeler plugin set the *id* attribute of a BridgeInterface object by concatenating the string "Port_"with the Port number, followed by the string "_ifIndex_" and the PortIfIndex, followed by the string "_RemIp_" and the RemoteAddress. The Python *prepid* function was applied to ensure uniqueness. An example would be *Port_13_ifIndex_2_RemIp_00_0C_41_9D_D3_81.*

The BRIDGE MIB provides values for:

- dot1TpPortInFrames (Counter32) .1.3.6.1.2.1.17.4.4.1.3
- dot1TpPortInFrames (Counter32) .1.3.6.1.2.1.17.4.4.1.4

These are performance counters for traffic seen on a transparent bridge port and they are indexed by port number. To be able to graph these values per-port, when an individual port is clicked on in the GUI, create a template with the name BridgeInterface for the /Devices/Network/Switch/BridgeMIB device class.

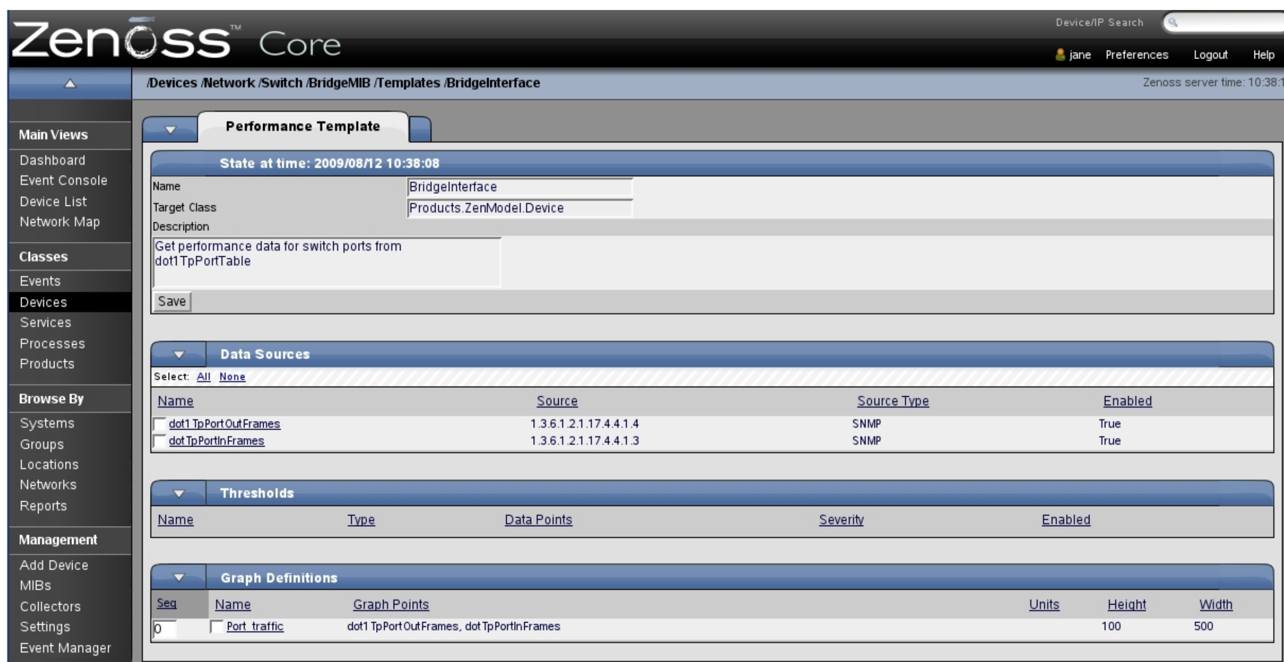

Figure 64: BridgeInterface template

There is no need to bind this template to any device or device class. To see performance data, simply click on a port under the *Bridge Interfaces* tab (remembering that it will generally take two SNMP polling intervals before data is displayed). Note in Figure 65 that the breadcrumb at the top shows the object's *id* attribute as the last element.

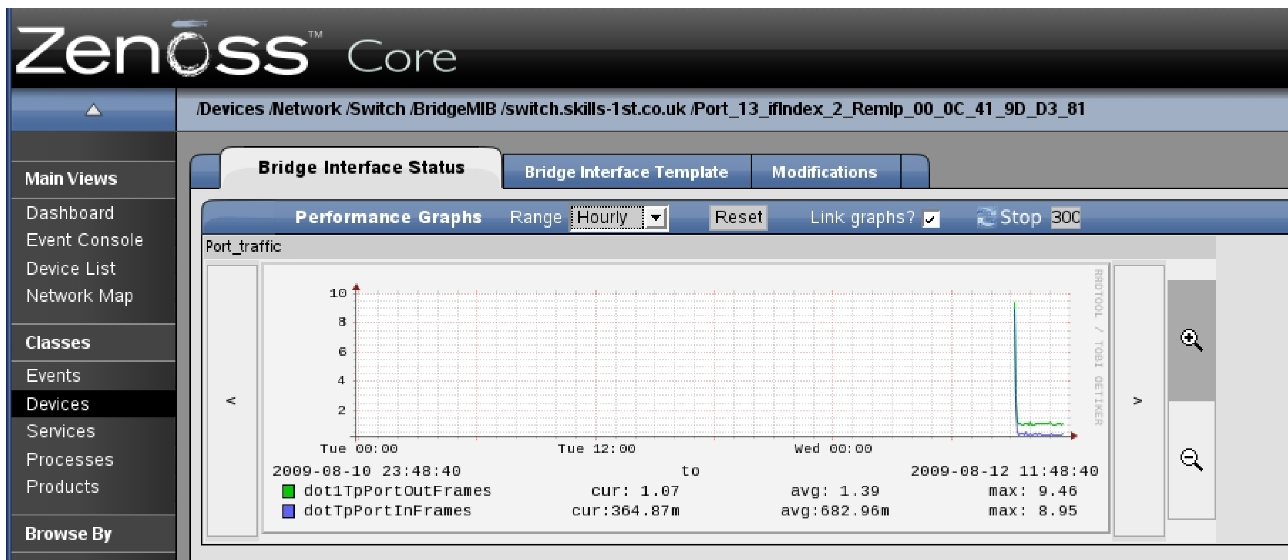

Figure 65: Performance graph for a specific switch port

The three tabs shown in Figure 65 were defined in the object class file BridgeInterface.py as shown in Figure 31 on page 41. The skins file to display performance graphs for a port is in *skins/ZenPacks.skills-1st.bridge/viewBridgeInterface.pt* shown in Figure 45 on page 61.

Remember to add the BridgeInterface performance template to the ZenPack when it is complete and to re-export the ZenPack.

6 Testing and debugging ZenPacks

The chances of getting a ZenPack with new source code, correct first time, is not high. This section offers some testing and debugging hints.

6.1 Testing

There may be three main areas where you have added code; object class files, modeler plugins and skins.

6.1.1 Testing new object class files

If you have created or changed object class files, you should always delete any discovered instances that use those files and rediscover them to ensure that any relationship changes are established correctly. You should certainly recycle **zenhub** and **zopectl** with:

- zenhub restart
- zopectl restart

Typically you will be doing initial testing with a single device so delete the device and use the *Add Device* menu to re-add it, ensuring that you specify your new device class in the *Device class path* dropdown. Adding the device runs *zendisc* which calls *zenmodeler*. You may see error messages in the discovery GUI. Usually they are quite good at pinpointing the problem to a particular line in a particular file. Watch out particularly for syntax errors in your code such as missing closing brackets, missing quotes or missing colons (:).

Another way to start testing object class files is to use the Zope interface to navigate to *http://zen241.class.example.org:8080/zport/dmd/manage* and then navigate down Devices/Network/Switch/BridgeMib/devices/<a specific device> and check that the BridgeInt relationship exists.

A classic error to make in Python files is to get white space indentation wrong. Python uses indentation to structure *if*, *while*, *for* and other structures; you must be consistent with the number of spaces used at each level of indentation.

6.1.2 Testing modeler plugins

If you have created or changed a modeler plugin, you need to restart **zenhub** and **zopectl**; typically you do **not** need to delete your test device and re-add it. It should be sufficient to simply use the *Manage -> Model Device* menu and watch the output.

Note the dialogue particularly to ensure that your modeler does at least attempt to run – the output will show what plugins are to be run.

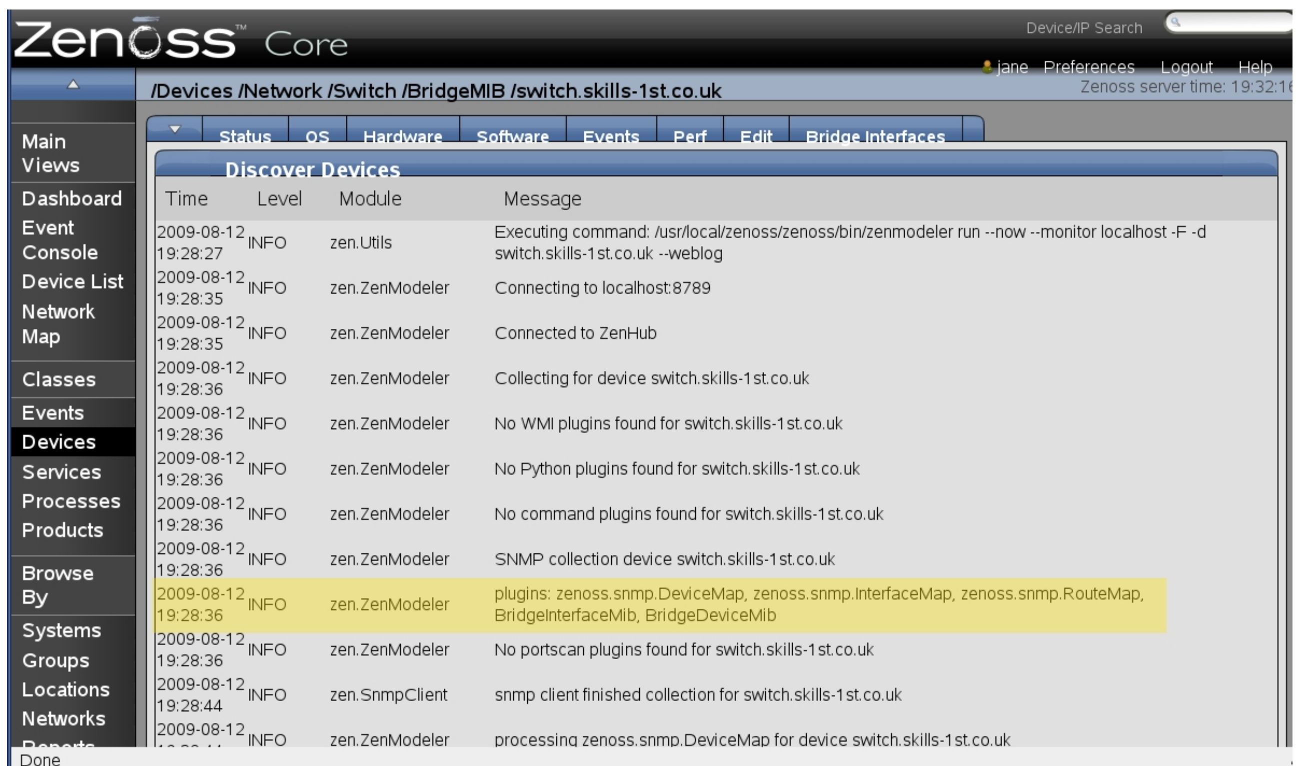

Figure 66: Output from Manage -> Model Device highlighting the plugins to be run

If your modeler doesn't appear on the plugins list it is probably a compilation error. Remember that you have created python source files (ending in *.py*); Zenoss will compile-on-demand to generate *.pyc* files. A good check is always to inspect the base Zenoss directory and the modeler/plugins directory to ensure that you have matching .pyc files for each of your .py files.

A good way to test for compilation errors is to use the *zendmd* utility to import the file in question.

```
zenoss@zen241: > zendmd
Welcome to the Zenoss dmd command shell!
'dmd' is bound to the DataRoot. 'zhelp()' to get a list of commands.
>>> from ZenPacks.skills1st.bridge.modeler.plugins import BridgeDeviceMib
>>> from ZenPacks.skills1st.bridge.modeler.plugins import BridgeDeviceMib
>>>
zenoss@zen241:~> zendmd
Welcome to the Zenoss dmd command shell!
'dmd' is bound to the DataRoot. 'zhelp()' to get a list of commands.
>>> from ZenPacks.skills1st.bridge.modeler.plugins import BridgeDeviceMib
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "/usr/local/zenoss/zenoss/local/jane/ZenPacks.skills1st.bridge/ZenPacks/skills1st/bridge/m
odeler/plugins/BridgeDeviceMib.py", line 32
    )
    ^
SyntaxError: invalid syntax
>>> 
```

Figure 67: zendmd dialogue showing successful compilation and unsuccessful compilation

The figure above shows a successful import – you simply receive a command prompt back. Note that you need to specify an **object** path to the Python source file, not a **file** path. The second zendmd dialogue shows a failed compilation (I removed a closing brace from line 32).

Note that, for some Python files you might also test compilation simply with:

```
python BridgeMib.py
```

however, you may get different compilation errors from this test as python on its own has no concept of the Zenoss environment or libraries whereas zendmd has, and python compilation may fail with unknown imports.

If the modeler runs but fails then hopefully you get a message in the GUI showing the modeler output. If there are insufficient clues here, try running zenmodeler standalone with full debugging turned on (*-v 10*):

```
zenmodeler run -v10 -d switch.skills-1st.co.uk
```

If you still can't see the problem, try putting log statements in the modeler plugin code to output intermediate data stages. Figure 68 highlights *log.warn* statements that output the results of the SNMP getdata and tabledata structures.

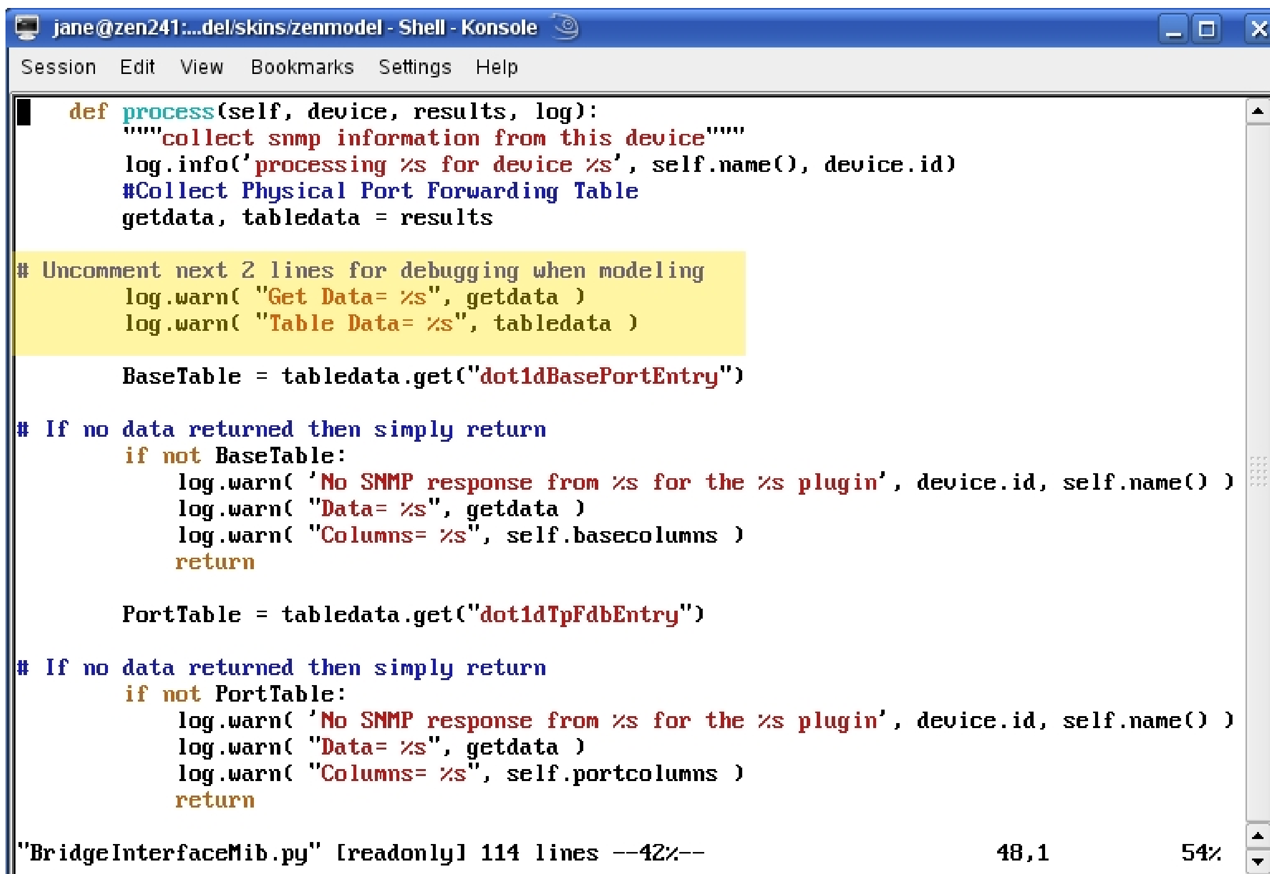

Figure 68: BridgeInterfaceMib.py code highlighting debugging logging

If you get really desperate, try the logging lines highlighted in Figure 69 to output all the attributes for an object instance; do not leave these lines uncommented once the problem is resolved.

```
jane@zen241:...del/skins/zenmodel - Shell - Konsole
Session  Edit  View  Bookmarks  Settings  Help
# dot1dTpFdbEntry table matches the Port number from the dot1dBasePortEntry

            om.PortIfIndex = -1
            for boid,bdata in BaseTable.items():
               if bdata['BasePort'] == om.Port:
                   om.PortIfIndex = bdata['BasePortIfIndex']

# prepId function ensures that results are all unique - will add _1, _2 etc to achieve this
            om.id = self.prepId("Port_" + str(om.Port) + "_ifIndex_" + str(om.PortIfIndex) +
 "_RemIp_" + str(om.RemoteAddress))

# For lots of debugging, uncomment next 2 lines
#            for key,value in om.__dict__.items():
#               log.warn("om key =  %s, om value = %s", key,value)

            rm.append(om)
        return rm

~
~
~
~
~
~
~
~
"BridgeInterfaceMib.py" [readonly] 114 lines --84%--                96,13          Bot
```

Figure 69: BridgeInterfaceMib .py highlighting debugging logging to output all the final object attributes

6.1.3 Testing skins files

If skins files have been created or changed, you generally only need to restart **zopectl** and then refresh the web page in the Zenoss GUI. If the code is incorrect a standard error page is shown and you can get more information by clicking the *View Error Details* link.

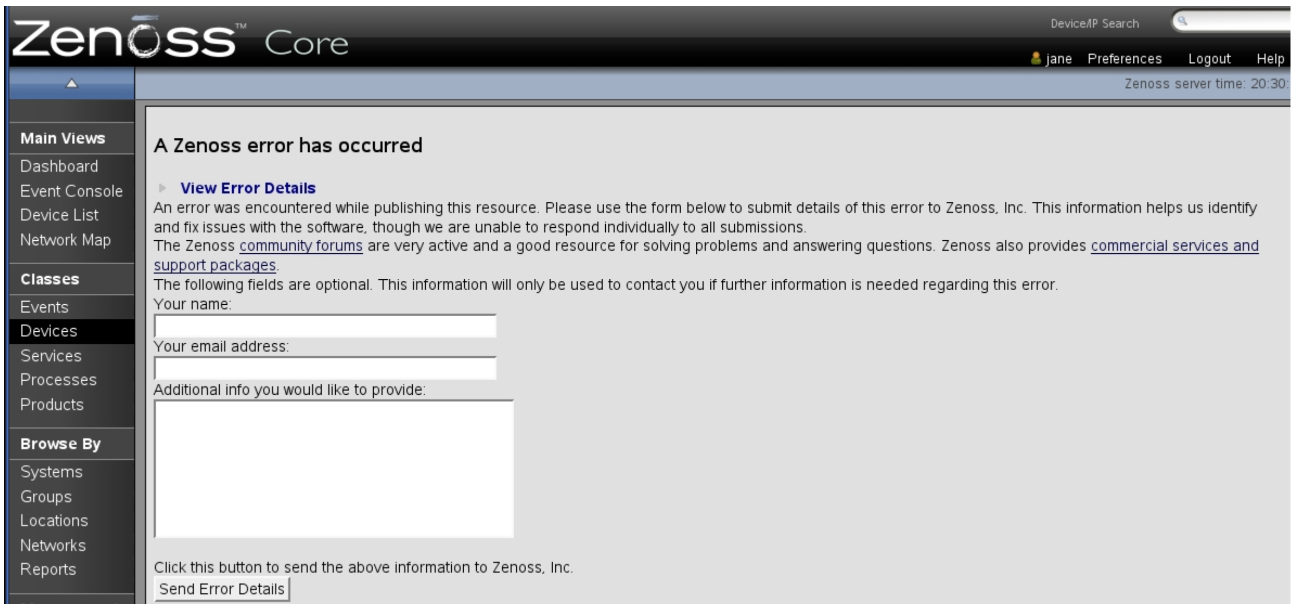

Figure 70: Standard error message for a faulty web page definition

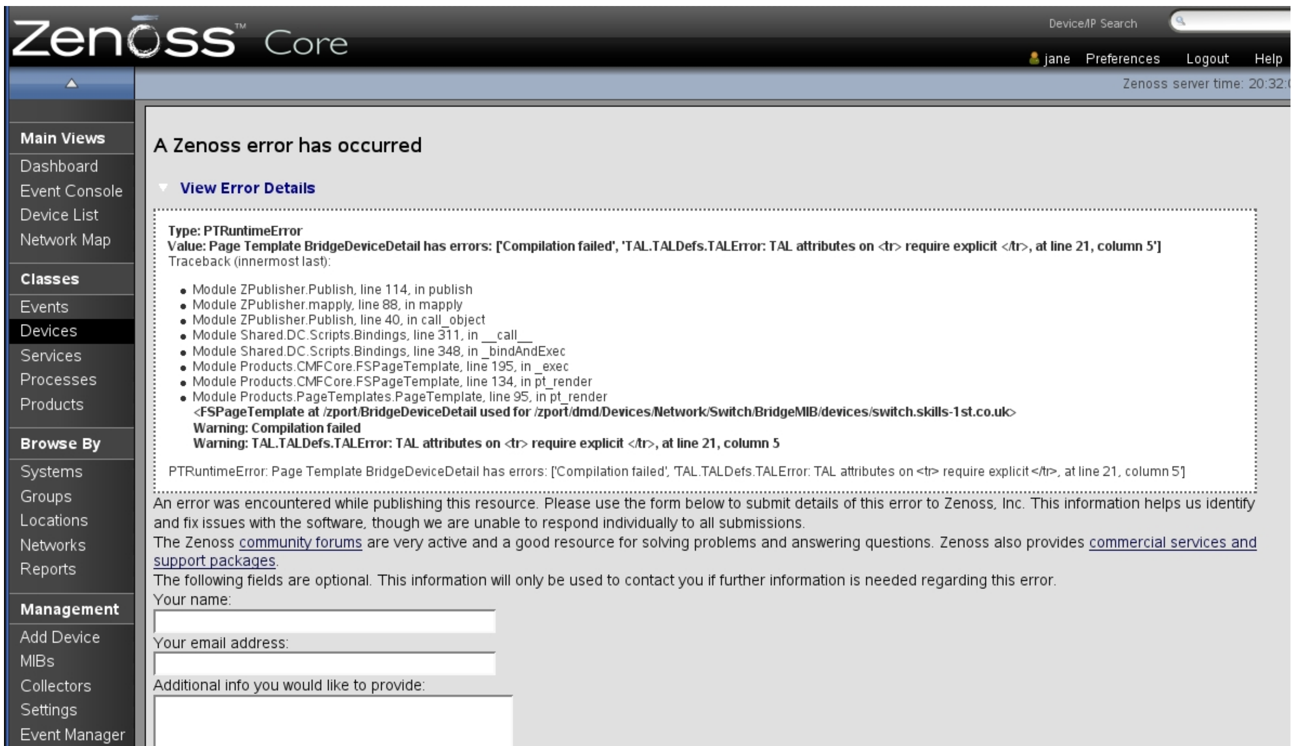

Figure 71: View Error Details for a faulty web page definition

Figure 71 shows the detailed error output. The file and line number at fault are documented (I had indeed commented out a closing </tr> at line 21 of the BridgeDeviceDetail.pt file). Simply fix the file, issue *zopectl restart* and refresh the web page.

Sometimes the web View Error Details page suggests something is wrong that is nowhere near anything you have recently changed. If this happens, try restarting the whole Zenoss system with:

```
zenoss stop
zenoss start
```

6.1.4 Debugging problems with performance data

There are several ways that performance data collection can fail:

- A template is created but not **bound** to a device. In this case, no attempt will be made to collect data. Go to the device's template page and use the table drop-down menu *Bind Templates* to check what is actually bound (remember that contained component templates, matching the component object class name, do **not** not binding – this happens automatically).
- Scalar MIB values need the trailing *.0*; otherwise no data will be collected.
- If SNMP community names configured in Zenoss do not match those in the target agents then you will get no SNMP data. Test with a simple snmpwalk command from a command line; for example:

```
snmpwalk -v 1 -c public switch.skills-1st.co.uk system
```

- If a template is correctly configured and bound but there are only one or two data values collected (counter values need at least two values before a point can be plotted as it is a rate-of-change measurement), you will see a graph with no data and the *cur, avg* and *max* values will have the value **nan**. This simply means graph points are not yet available; another snmp polling interval usually fixes this issue.
- For component, contained device templates collecting tables of SNMP data, the instance may be the issue. Increasing the logging level for zenperfsnmp may help diagnose this.

Templates collect data into Round Robin Database (rrd) files, held under $ZENHOME/perf/Devices with a separate subdirectory for each device and each device may have subdirectories for components such as *os* or *BridgeInt* (ie. the **relationship** name of the contained device).

Always check that rrd files exist. Templates for devices have the format:

```
<datasource name>_<datapoint name>.rrd
```

jane@zen241:...del/skins/zenmodel - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
-rw-r--r-- 1 zenoss zenoss 35296 2009-08-13 10:04 sysUpTime_sysUpTime.rrd
zenoss@zen241:/usr/local/zenoss/zenoss/perf/Devices>
zenoss@zen241:/usr/local/zenoss/zenoss/perf/Devices>
zenoss@zen241:/usr/local/zenoss/zenoss/perf/Devices>
zenoss@zen241:/usr/local/zenoss/zenoss/perf/Devices> ls -l
total 60
drwxr-x--- 3 zenoss zenoss 4096 2009-05-28 15:16 adsl2.skills-1st.co.uk
drwxr-x--- 3 zenoss zenoss 4096 2009-06-30 17:14 bino.skills-1st.co.uk
drwxr-x--- 3 zenoss zenoss 4096 2009-08-06 05:17 deodar.skills-1st.co.uk
drwxr-x--- 3 zenoss zenoss 4096 2009-06-22 16:12 group-100-linux.class.example.org
drwxr-x--- 3 zenoss zenoss 4096 2009-06-30 15:46 group-100-r1.class.example.org
drwxr-x--- 3 zenoss zenoss 4096 2009-06-22 13:27 group-100-r2.class.example.org
drwxr-x--- 3 zenoss zenoss 4096 2009-06-22 13:32 group-100-r3.class.example.org
drwxr-x--- 4 zenoss zenoss 4096 2009-08-04 12:05 group-100-s1.class.example.org
drwxr-x--- 4 zenoss zenoss 4096 2009-08-12 09:46 group-100-s2.class.example.org
drwxr-x--- 3 zenoss zenoss 4096 2009-05-28 15:16 hp7410.skills-1st.co.uk
drwxr-x--- 3 zenoss zenoss 4096 2009-06-30 15:36 server.class.example.org
drwxr-x--- 4 zenoss zenoss 4096 2009-08-12 09:41 switch.skills-1st.co.uk
drwxr-x--- 3 zenoss zenoss 4096 2009-08-12 19:16 taplow-20.skills-1st.co.uk
drwxr-x--- 3 zenoss zenoss 4096 2009-08-12 18:17 team1itm.class.example.org
drwxr-x--- 3 zenoss zenoss 4096 2009-07-06 10:46 zen241.class.example.org
zenoss@zen241:/usr/local/zenoss/zenoss/perf/Devices> ls -l switch.skills-1st.co.uk/
total 116
drwxr-x--- 9 zenoss zenoss  4096 2009-07-30 21:04 BridgeInt
-rw-r--r-- 1 zenoss zenoss 35296 2009-08-13 10:04 dot1dStpTimeSinceTopologyChange_dot1dStpTimeSinceTopologyChange.rrd
-rw-r--r-- 1 zenoss zenoss 35296 2009-08-13 10:04 dot1dStpTopChanges_dot1dStpTopChanges.rrd
drwxr-x--- 3 zenoss zenoss  4096 2009-07-09 12:08 os
-rw-r--r-- 1 zenoss zenoss 35296 2009-08-13 10:04 sysUpTime_sysUpTime.rrd
zenoss@zen241:/usr/local/zenoss/zenoss/perf/Devices>
```

Shell

Figure 72: Directories for performance files for devices

If you see graphs that have no data at all, this generally means that a template is bound but there is no rrd file, as shown in Figure 73.

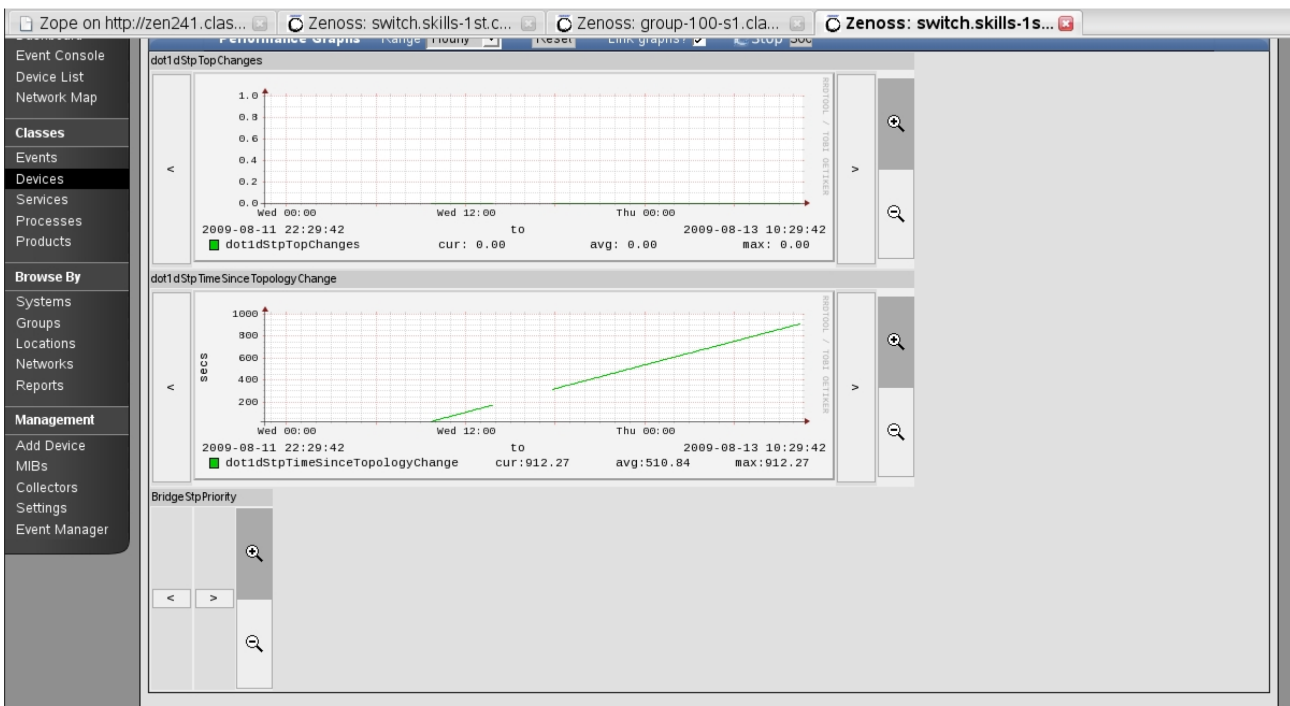

Figure 73: The empty graph at the bottom suggests that a template is bound but no data has been collected

Note that when you configure data sources in a template, there is a test button that you can use to specify a device known to Zenoss; however, the test that is run, strictly, is an *snmpwalk* whereas the zenperfsnmp daemon is more likely to issue an *snmpget,* so the test button can disguise problems with instances. Note that versions of Zenoss prior to 2.4 did not always implement the Test button correctly.

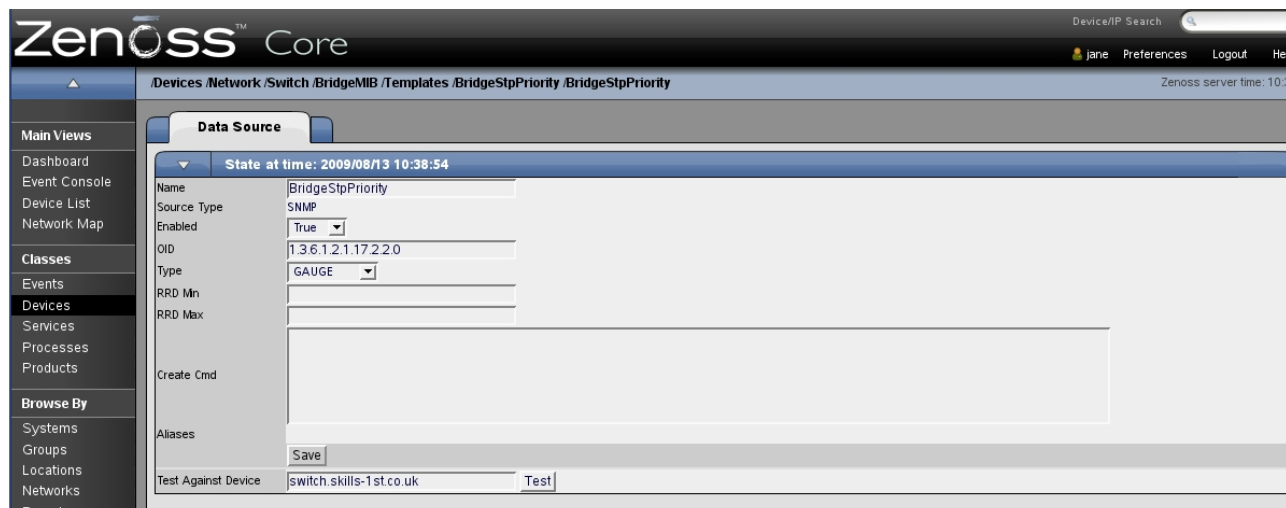

Figure 74: Using the test button from the Data Source configuration dialogue

6.1.5 General testing and debugging hints and tips

There are two two general areas for debugging help. Zenoss logfiles are all held under *$ZENHOME/log*. By default they have an *Info* level of logging but this can be increased to *Debug* to provide lots more data. When the problem is resolved, the original logging level should be restored.

Daemon log files and their configuration can be inspected from the *Settings* menu under the *Daemons* tab. From Zenoss 2.4, the *edit config* link offers a separate page to choose configuration options (previous versions simply took you to an editable copy of the configuration file). To increase the debug level, change the *logseverity* to *Debug*. If you check the configuration file for this daemon in *$ZENHOME/etc* you will see a line:

```
logseverity   10
```

Any changes to a daemon's configuration file requires a restart of the daemon, either through the GUI or using *<daemon> restart* from a command line.

In addition to checking specific Zenoss daemon files like zenmodeler.log or zenperfsnmp.log, it is always worth also checking **zenhub.log** and **event.log.**

The second general debugging tool is **zendmd**. This is a Python interpretive environment provided by Zenoss that already understands some of the Zenoss object hierarchy. It is an excellent “sandpit” to test out bits of Python and to query Zenoss

 © Skills 1st Ltd

objects and their attributes and methods. Several examples have already been demonstrated throughout this document.

When weird things happen that really make no sense at all, try recycling the whole Zenoss system with a:

```
zenoss stop
zenoss start
```

7 Conclusions

ZenPacks are a powerful and flexible way to extend core Zenoss capability. Development mode provides a simple method to achieve simple ZenPacks. Source mode ZenPacks require more understanding of Zenoss internals, SNMP and of Python, but anything is possible.

The Bridge MIB ZenPack will be available from the Zenoss ZenPacks website - http://www.zenoss.com/community/projects/zenpacks/ . The source code for the ZenPack is available, with this document, at http://www.skills-1st.co.uk/papers/jane/zenpacks/ .

References

1. Zenoss Developer's Guide 2.4 - http://www.zenoss.com/community/docs
2. Zenoss Administration Guide 2.4 - http://www.zenoss.com/community/docs
3. Zenoss Extended Monitoring Guide 2.4 for documentation on Core and Enterprise ZenPacks - http://www.zenoss.com/community/docs/
4. Zenoss ZenPack FAQ at

   http://www.zenoss.com/community/projects/zenpacks/zenpack-documentation/zenpack-faq
5. Zenoss ZenPacks site at http://www.zenoss.com/community/projects/zenpacks/
6. Zenoss-ZenPacks forum at http://forums.zenoss.com/viewforum.php?f=6
7. Zenoss community developer site wiki at http://community.zenoss.org/trac-zenpacks/wiki , look especially at "Instructions for creating an ssh ZenPack" and "Diving into the device model".
8. "Custom ZenPacks rough guide" contributed by ***blacks*** to the Zenoss forum at http://www.zenoss.com/community/wiki/user-contributed/CustomZenPackRoughGuide/
9. Zenoss download site - http://www.zenoss.com/download/links/
10. net-SNMP SNMP agent from http://www.net-snmp.org/
11. BRIDGE MIB, RFC 1493 - http://www.ietf.org/rfc/rfc1493.txt
12. oidview online website for viewing MIBs such as the BRIDGE MIB - http://www.oidview.com/mibs/0/BRIDGE-MIB.html
13. "Learning Python" by Mark Lutz, published by O'Reilly
14. Zope web application server information from http://www.zope.org/WhatIsZope
15. Zope Page Templates Reference - http://docs.zope.org/zope2/zope2book/source/AppendixC.html
16.

Acknowledgements

Several people have contributed either actively or passively to this paper:

- "blacks" on the Zenoss forum for his Custom ZenPack Rough Guide that got me started. The original work for this was submitted by Zach Davis.
- Danny Deng who sent me his ZenPack samples and explanations
- George Fakhri for his blog post on "How to create a ZenPack.."

 © Skills 1st Ltd