

Nets Developers Guide

Copyright (C) 1999-2005
Mike McCauley and Hugh Irvine

Developers guide to customizing and
extending the Nets network inventory and
management system.
For Nets Revision 2.5

1.0 Introduction

Nets is a graphical source code product providing a flexible and extensible environment for maintaining essential network inventory, configuration and cost information in a platform and database independent manner.

Nets provides a physical model of the network, from which it is simple and straightforward to derive and display any number of textual, logical or graphical views.

Nets is designed by network engineering professionals to address the common problems encountered in network design, management and operation.

In keeping with its flexible and extensible design, Nets provides a number of ways to customize and extend its functionality. You can add and alter functions specifically for your own site, and you can also create reusable packages of functions and features that can be distributed and installed at other Nets sites.

This document describes the different methods that can be used to customize and extend Nets.

Readers of this document are expected to have some knowledge of system administration, and to have an understanding of standard Nets functionality, SQL databases and the Perl programming language.

2.0 Overview

Nets has been designed to be easily extended, enhanced, customized and modified, both to suit your own special needs, and also to distribute changes to other Nets users. There is a community of Nets users who contribute packages of code and functions, and all Nets users are encouraged to contribute interesting and useful enhancements.

There are a range of ways that Nets can be modified, and you may use one or more methods depending on the type of changes or enhancements you need to make:

- Because Nets is a source code product, it is always possible to modify the source code directly. See Section 5.0 on page 6 for a detailed description of the organization of the source code, and some common places where enhancements and changes may be made.
- Plug-ins are modules of Perl code that are automatically loaded into the Nets program at start-up. A Plug-in can contain any Perl code, and has full access to all the data Nets data structures and code, just like the native Nets code. Plug-ins are an ideal way to integrate new functions and features into Nets without modifying the standard Nets code. See Section 7.0 on page 17 for a detailed description of Plug-ins.
- Hooks provide a mechanism to let your own perl code get control during normal Nets processing. Hooks can be established by a Plug-in, and can be used to provide special processing of database queries and updates, logging, and at other times. See Section 6.0 on page 15 for more details.
- You can add to and update the contents of the Nets SQL database using Nets Load Files. When a Nets Load File is imported, it can add new records to any database table, or modify existing ones. Special features in the file syntax allow you to easily establish relations between Nets objects in the database. See Section 8.0 on page 19 for more details.
- Templates allow Nets users to easily create groups of interrelated Nets Objects, typically in order to quickly construct all the parts of a complicated Device or Interface. See Section 9.0 on page 21 for more details
- Packages allow developers to provide new Nets functionality in a single, easy to distribute bundle. Nets users can easily load packages into their Nets system, to add the features that are part of the Package. Packages can contain any or all of the other Nets extension and enhancement features described in this document. See Section 10.0 on page 22 for more details
- Nets provides a simple scripting language that allows some Nets actions and features to be triggered. While such scripts do not have the full power of Plug-ins (which have the power of Perl, plus all the Nets API and data structures to work with), they can be used to easily add simple actions and features to Nets. See Section 11.0 on page 26 for more details.

3.0 The Nets distribution

The standard Nets distribution includes a number of files and directories, and this section describes the major directories included and how they are used.

3.1 Main distribution directory

When a Nets distribution file is unpacked (using tar/gzip or WinZip, depending on your platform), it creates a single directory containing the entire Nets software tree. The top of this distribution tree will be named something like `Nets-2.1`, depending on exactly which version of Nets you have. This top level directory contains the Nets executable programs, some data files, and a number of important sub-directories described below.

The executable programs include `netsmain.pl` (the main Nets user program), plus some utility programs, `netscreate.pl`, `netsload.pl`, `netsdump.pl` and `netspkgininstall.pl`. When Nets is installed, these executable programs are copied to the standard Perl binary directory (typically `/usr/local/bin` on Unix, or `C:\Perl\bin` on Windows).

In the following sections, `<NetsDir>` means the directory at the top of your Nets distribution.

3.2 `<NetsDir>/Nets`

This directory includes all the Nets Perl modules required by the Nets programs mentioned above. Each module has a `.pm` extension. When Nets is installed, these modules are copied to the standard Perl site library directory (typically `/usr/local/lib/perl5/site_perl` on Unix, or `C:\Perl\site` on Windows). You can add your own modules to Nets by adding them to this directory and reinstalling Nets.

3.3 `<NetsDir>/images/icons`

Contains the standard Nets icons, which are using on Nets Drawings and other parts of the Nets user interface.

3.4 `<NetsDir>/images/drawings`

Contains the background images for Nets Drawings

3.5 `<NetsDir>/templates`

Contains the standard templates, used for creating collections of interrelated Nets Objects.

3.6 `<NetsDir>/plugins`

Contains the standard Plug-ins provided as part of Nets.

3.7 `<NetsDir>/scripts`

Contains the standard Nets Scripting Language scripts provided with Nets.

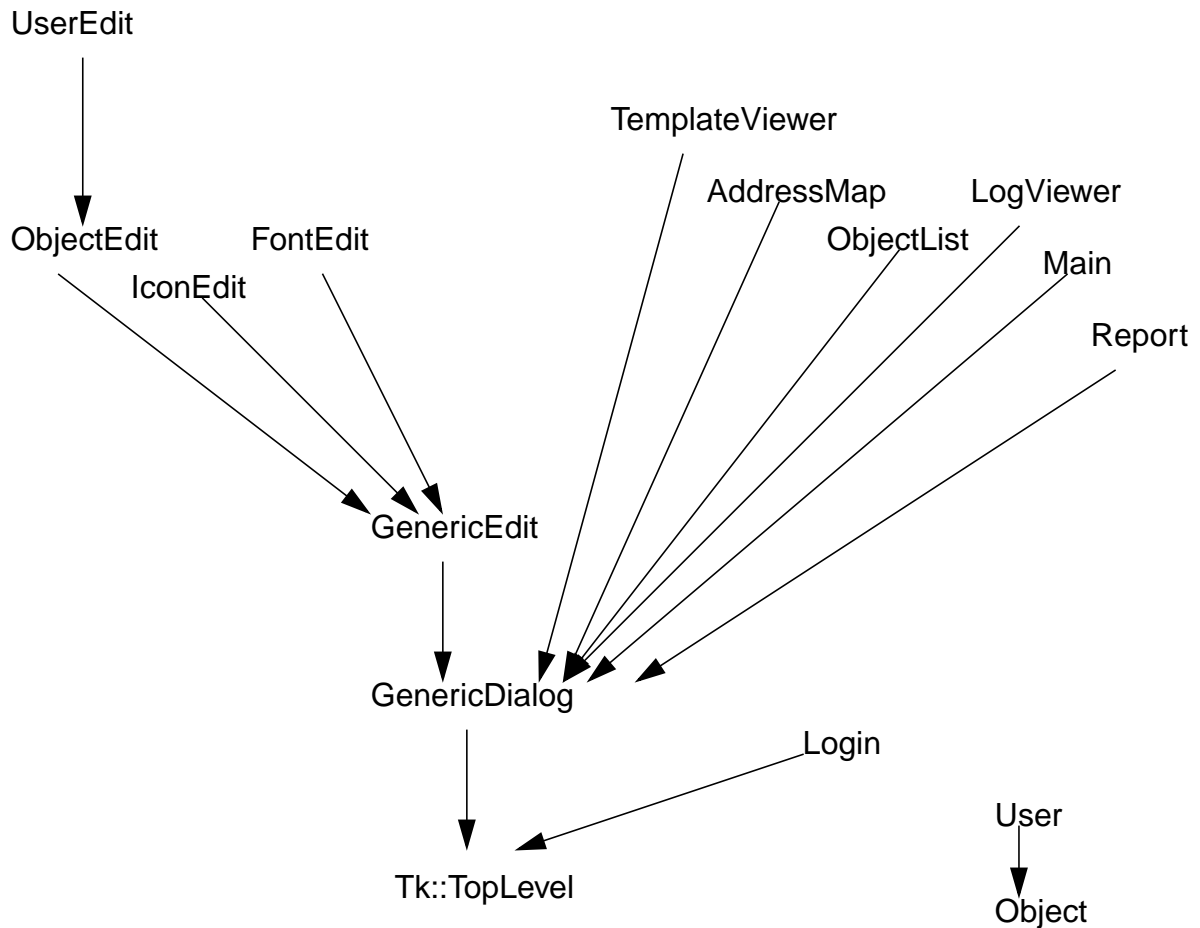
3.8 `<NetsDir>/goodies`

A collection of contributed and unsupported files that developers may be interested in. See the README file for a brief description of each file.

4.0 Object Hierarchy

Nets has been designed and implemented using modern Object Oriented software technology. This design allows Nets developers to easily use and extend the core features of Nets. Figure 1 on page 4 shows the core Nets object hierarchy.

FIGURE 1. Nets Object Hierarchy



The core Nets product includes the Perl classes shown below. In general, each Nets Perl class is defined by a Perl package (not to be confused with a Nets package), using conventional Perl object packages. You can add your own classes using Plug-ins, or by modifying the Nets code.

4.1 DBSQL

Provides a range of low level SQL database access routines for doing queries, inserts, updates and deletes. `$main::db` refers to the (usually) one instance of the DBSQL class.

4.2 DBUtil

Provides a range of routines for managing the contents of the SQL tables as described by `Schema.pm`. Uses the services of DBSQL to interact with the SQL database.

4.3 Object

This class provides “persistent objects”, used to implement Nets Objects such as DEVICE, LINK, NETSUSER etc. Uses the services of DBUtil to save objects to and from the SQL database.

4.4 User

Subclass of Object used for storing information about Nets Users, including Preferences and Permissions.

4.5 GenericDialog

Provides a generalized Tk window class, with a number of common routines and data structures used by most Nets windows, including:

- Menu bar, with common standard menus
- A status bar, for user messages
- Configuring windows
- Storing and reloading window descriptions
- Window history, for implementing Forward and Back functions
- Message Box
- Running tools

4.6 GenericEdit

Subclass of GenericDialog adds features required by a range of object editors, such as OK, Cancel and Help buttons, object saving, and list of available objects.

4.7 ObjectEdit

Subclass of GenericEdit provides object editing services for most low-level Nets objects. Objects are displayed according to the user interface description for the Object class as described in `UIDesc.pm`. User permissions are observed.

4.8 IconEdit

Subclass of GenericEdit that provides preview and selection of icons from the `<Nets-Dir>/images/icons` directory.

4.9 FontEdit

Subclass of GenericEdit that provides preview and selection of fonts. Font names are represented with a platform independent font naming scheme.

4.10 UserEdit

Subclass of ObjectEdit that provides a specialized user interface for editing Nets User objects.

4.11 ObjectList

Subclass of GenericDialog that provides listing and searching for Nets objects. The displayed fields, and searchable fields are described in UIDesc.pm.

4.12 TemplateViewer

Subclass of GenericDialog that provides selection and loading of Templates, as well as click-through to the objects created by the Template.

4.13 AddressMap

Subclass of GenericDialog that provides a window for visualizing IPV4 address usage.

4.14 LogViewer

Subclass of GenericDialog that provides a window for viewing Nets log messages in real time. (This class provides a working example of registering and deregistering Hooks).

4.15 Main

Subclass of GenericDialog implements the Nets main window. This window is always created when Nets starts up, and is always present.

4.16 Report

Subclass of GenericDialog that provides a window for selecting and viewing reports.

5.0 Source Code

Nets is a source code product, and is provided with all (or most, depending on your license) of the source code. One easy way to modify the operation of Nets is to modify the source code directly. While this is easy, it is discouraged, since whenever a new version of Nets is released, you will need to migrate your changes from the old version to the new version. In fact the preferred way to add or change Nets functions is to use a Plug-in (see Section 7.0 on page 17).

However, if you really do wish to alter the source code, the following sections will provide some information about where and how you should make your changes. The pre-

ferred way of changing the source code is to edit the source file in the Nets distribution, and then reinstall Nets, so that the changes are moved to the public Perl binary and module directories.

Migrating your source code changes from Nets version to version can be made easier with the unix tools `diff(1)` and `patch(1)`.

5.1 Schema.pm

This file contains the definitions of all the SQL tables used by Nets. The tables are described in a data structure, and in a database independent way. Changing `Schema.pm` allows you to easily add, change and remove tables and columns from your Nets database.

You might, for example wish to add new columns to a table in order to store some site-specific information that your organization needs. Note that you would normally also need to add something to the `ObjectEdit` user interface for that object, by modifying `UIDesc.pm` (see Section 5.2 on page 9).

The database schema described in `Schema.pm` is used by Nets to load and save Nets Objects. It is also used by `netscreate.pl` when it creates (or recreates) a database table. You should note that altering `Schema.pm` will not, on its own, change the table in the database, it merely changes what tables and columns Nets will refer to. You might need to drop and recreate a table, or perhaps add or drop columns by hand, if dropping a table is not possible. The Nets utilities can provide some help with this. For example, you could dump your entire database with `netsdump.pl`, alter the schema, recreate the tables with `netscreate.pl -dropfirst`, then reload the data with `netsload.pl`.

If you are just adding a new table, it should be sufficient to just do `netscreate.pl`. By default, `netscreate.pl` does not drop any existing tables, so the new table will be created without affecting any existing table (Note, you will get a number of warning messages as `netscreate.pl` tries to create all the already existing tables).

`Schema.pm` defines the data structure `%Nets::schema`. This is a perl hash, containing an entry for each table used by Nets. Each entry defines the columns, keys and name for that table. Here is a typical example, which defines a table for network addresses.

```
%Nets::schema =
(
  ADDRESS => {
    'title' => 'Address',
    'description' => 'One record for each address',
    'columns' => {
      ADDRESSTYPE      => [ 'fk', 'ADDRESSTYPE.ID' ],
      DESCRIPTION      => [ 'varchar', 200 ],
      DNSNAME           => [ 'varchar', 50 ],
      ID                => [ 'id' ],
      NAME              => [ 'varchar', 50 ],
      NETMASK           => [ 'varchar', 50 ],
      OVERSION          => [ 'version' ],
    },
  },
),
```

```

        'primary_key' => [ 'ID' ],
    },

```

As can be seen, the hash entry for ADDRESS is in fact an anonymous hash with entries for title, description, columns and primary_key. The following types are supported:

TABLE 1. Database schema entry types

Entry type	Contents
title	Informative natural language name for this table. e.g.: 'title' => 'Address'
description	Informative natural language description of this table. e.g.: 'description' => 'One record for each address'
columns	Anonymous hash, each entry and array of column definition entries, see Table 2 on page 8 e.g.: 'columns' => { ADDRESSTYPE => ['fk', 'ADDRESSTYPE.ID'], DESCRIPTION => ['varchar', 200],
primary_key	Optional array of column names. The primary key will be formed from all the columns named. e.g.: 'primary_key' => ['ID'],
unique	Optional array of arrays of column names. A unique index will be created for each set of column names. e.g.: 'unique' => [['NAME', 'NETMASK'], ['DNSNAME', 'DESCRIPTION']], will create two unique indexes, one on NAME and NETMASK, and the other unique index on DNSNAME and DESCRIPTION.
index	Optional array of arrays of column names. A (non-unique) index will be created for each set of column names. e.g.: 'unique' => [['NAME', 'NETMASK'], ['DNSNAME', 'DESCRIPTION']], will create two indexes, one on NAME and NETMASK, and the other on DNSNAME and DESCRIPTION.

TABLE 2. Database column definition types

Type	Field 1	Field 2	Remarks
id	Column name		Enforces a unique integer ID for each row in the table. When a row is inserted, a new unique ID will be allocated from the OBJECTTYPE table.
version	“		Enforces an automatically incremented version number for the row. The version number can be used to detect editing collisions. It is automatically incremented on each update done through DBUtil::update. All standard Nets objects utilize this feature.

TABLE 2. Database column definition types

Type	Field 1	Field 2	Remarks
fk	“	TAB.COLUMN	A foreign key to another Nets table. The contents of this column matches a row with the given column name in named table.
int	“		An integer value
varchar	“	chars	A Variable width text field. Max length is given by Field 2.
float	“		A Floating point number
datetime	“		A date/time. Native database date/time types are not used by Nets. All Nets date/times are represented as seconds since Jan 1 1970 (i.e. Unix epoch time)

5.2 UIDesc.pm

This file defines the user interface for each type of Nets object. It is used by the ObjectEdit and ObjectList classes to construct the right type of user interface for each type of Nets object. In general terms, for each type of object, it specifies which database columns are to be displayed to the user, and how to display them. Many features of the user interface can be customized and controlled by changing the entries in UIDesc.pm

UIDesc.pm defines 2 hashes: %Nets::UIDesc::editors, and %Nets::UIDesc::listers. Editors defines the user interface for editing objects, used by ObjectEdit.pm. Listers defines the user interface for listing objects, used by ObjectList.pm. If there is no entry for a given object type, then no editing window or search/list window (as the case may be) can be created.

5.2.1 %Nets::UIDesc::editors

This hash defines the user interface for editing objects. It is used by ObjectEdit.pm to construct the user interface appropriate to each type of Nets object.

Each entry in editors is an anonymous hash, with each entry in that hash describing some element of the user interface. Not all elements are required, some being optional. Here is a typical example, which defines the object editing user interface for ADDRESS objects:

```
%editors =
(
ADDRESS => {
title => 'Address',
index_field => ['text', 'NAME'],
main_window =>
[
['entry', 'NAME', 'Address', 'This address', undef, 50],
['entry', 'DNSNAME', 'DNS Name', 'The DNS name', undef, 50],
['entry', 'NETMASK', 'Netmask', 'The netmask', undef, 50],
['fkmenu', 'ADDRESSTYPE', 'Address Family', 'Address family'],
['entry', 'DESCRIPTION', 'Description', 'Description', undef,
50],
],
],
```

```

defaults => {
  ADDRESSTYPE => 2, # IP V4
  NETMASK     => '255.255.255.0',
},
validator => \&validateAddressEditor,
tools =>
[
  ['Edit INTERFACES with this ADDRESS',
   'Nets::Session::show',
   type => 'INTERFACE',
   where => 'ADDRESS=%{context_id}',
   title => 'Interfaces with Address'],
  ['Show Address Map',
   'Nets::Session::show',
   class => 'Nets::AddressMap'],
],
view_permissions => ('VIEW_NETWORK'),
edit_permissions => ('EDIT_NETWORK'),
sortby => 'IP address',
},

```

TABLE 3. %Nets::UIDesc::editors entry types

Entry type	Comments
title	Natural language name, used in the title bar of the editor.
index_field	Specifies which column(s) from each object are to be shown in the object list on the left side of the editor window. Also specifies the display format for each column.
main_window	A hash that specifies which column(s) from the currently selected object will be displayed in the main panel on the right hand side of the editing window. See Table 4 on page 11 for details of the format of the entries in this hash.
panels	Optional hash that specifies a number of sub-panels to display as notebook tabs under the main panel on the right hand side of the editor. Within a panel, the same format as main_window is supported, See Table 4 on page 11 for details of the format of the entries in this hash.
defaults	Optional hash giving default values for each column. These defaults will be applied when a new object is created in the Editor.
validator	Optional reference to a function that will be called to validate the contents of the editor before the object is saved to the database. If the object validates OK, returns undef, else returns a string that will be displayed to the user.
tools	Optional array of tool definitions. Each definition results in an entry in the Tools menu for this object. See Table 6 on page 12.
view_permissions	Optional array of permission names. In order to view the contents of an object of this type, the current Nets user must have one of the permissions mentioned here, or else VIEW_ANYTHING or VIEW_<objectname> permissions.

TABLE 3. %Nets::UIDesc::editors entry types

Entry type	Comments
edit_permissions	Optional array of permission names. In order to change or delete the contents of an object of this type, the current Nets user must have one of the permissions mentioned here, or else EDIT_ANYTHING or EDIT_<objectname> permissions.
sortby	Optional string, specifies the initial sort order to use in the object list on the left side of the editor. May be one of: 'Natural', 'Id', 'Alphabetic', 'Alphabetic (reverse)', 'Numeric' or 'IP address'. Defaults to 'Natural', ie the natural order returned by the database.

TABLE 4. %Nets::UIDesc::editors field definitions

Field	Meaning
0	The type of user interface widget to use to display this column. May be one of entry, datetime, ipv4address, text, static, timestamp, latlong, int, float, currency, menu, colour, font, icon, fkselect, fkmenu. See Table 5 on page 11 for details.
1	The name of the column that contains the data to display.
2	The prompt, or field name.
3	Short help text that appears in a help balloon over this field.
4	For 'menu', a reference to a function that returns a list of menu entries, use to populate the menu. For fkmenu and fkselect, a list of additional columns to show in the menu.
5	entry, datetime and ipv4address, text, int, float, currency, the width of the text entry widget in characters.
6	For text, the height of the text entry widget in rows.

TABLE 5. %Nets::UIDesc::editors interface widget type

Widget type	Description
entry	A simple single-line text fields.
text	Multi-line text field.
datetime	Date/time field. Presents an editable text field that accepts any of the date/tine formats permitted by Nets::Util::parseDate.

TABLE 5. %Nets::UIDesc::editors interface widget type

Widget type	Description
ipv4address	An editable field that requires a dotted quad IP V4 address (e.g. 203.63.154.1).
static	Non-editable simple text.
timestamp	Non-editable date/time field.
latlong	An editable field that requires the input to look like a latitude or longitude (e.g. 110.5E, or 110 30.0E).
int	An integer.
float	A floating point number.
currency	A money field, formatted according to \$Nets::config{CurrencyFormat}.
menu	A (usually short) menu of text items. The menu is passed in as field 4.
colour	Presents a button that pops up a colour selector window, and shows the currently selected colour.
font	Presents a button that pops up a font selector window, and shows the currently selected font.
icon	Presents a button that pops up an icon selector window, and shows the currently selected icon.

TABLE 6. %Nets::Tools::editors tools definitions

fkselect	Presents a label showing the currently selected foreign key, plus a button that allows you to choose a new one.
fkmenu	Presents an option menu showing the currently selected foreign key, plus a button that allows you to browse all the options. This is really only suitable for lists of less than 30 or so items.
fkcombo	Presents a popup scrolling list (like a Windows combo-box) for choosing a foreign key, plus a button that allows you to browse all the options. This is suitable for large numbers of choices.

You should note that it is possible to alter the %Nets::UIDesc::editors hash without editing UIDesc.pm. For example, from within a Plug-in, you could add a new user interface description for a new type of object with something like this:

```

$Nets::UIDesc::editors{'MYNEWOBJECT'} =
{
  title => 'My New Object',
  index_field => ['text', 'NAME'],
  main_window =>
  .....
};

```

5.2.2 %Nets::UIDesc::listers

This hash defines the user interface for editing objects. It is used by ObjectList.pm to construct the user interface appropriate to each type of Nets object.

Each entry in listers is an anonymous hash, with each entry in that hash describing some element of the user interface. Not all elements are required, some being optional. Here is a typical example, which defines the object search/list user interface for ADDRESS objects:

```
%listers =
(
  ADDRESS => {
    title => 'Address',
    main_window =>
    [
      ['text', 'NAME', 'Address', 'Address', 20],
      ['text', 'DNSNAME', 'DNS Name', 'DNS name', 20],
      ['text', 'DESCRIPTION', 'Description', 'Brief description
        of the usage of this address', 20],
      ['text', 'NETMASK', 'Netmask', 'The network netmask', 20],
      ['fkmenu', 'ADDRESSTYPE', 'Address Family', 'The type of
        address family for this address', 10],
    ],
  },
  .....
```

TABLE 7. %Nets::UIDesc::listers entry types

Entry type	Comments
title	Natural language name, used in the title bar of the editor.
main_window	A hash that specifies which column(s) from the currently selected object can be searched on, and which will be displayed in the list at the bottom of the window. See Table 8 on page 13 for details of the format of the entries in this hash.

TABLE 8. %Nets::UIDesc::listers field definitions

Field	Meaning
0	The type of user interface widget to use to display this column. May be one of datetime, text, int, float, currency, fkmenu. See Table 9 on page 14 for details.
1	The name of the column that contains the data to display.
2	The prompt, or field name.
3	Short help text that appears in a help balloon over this field.

TABLE 8. %Nets::UIDesc::listers field definitions

Field	Meaning
4	For 'menu', a reference to a function that returns a list of menu entries, use to populate the menu. For fkmnu and fkselect, a list of additional columns to show in the menu.
5	entry, datetime and ipv4address, text, int, float, currency, the width of the text entry widget in characters.
6	For text, the height of the text entry widget in rows.

TABLE 9. %Nets::UIDesc::lister interface widget type

Widget type	Description
text	Simple text field.
datetime	Date/time field. Presents an editable text field that accepts any of the date/tine formats permitted by Nets::Util::parseDate.
int	An integer.
float	A floating point number.
currency	A money field, formatted according to \$Nets::config{CurrencyFormat}.
fkmnu	Presents an option menu showing the currently selected foreign key, plus a button that allows you to choose a new one.

5.3 Site.pm

This file contains the initial values of various configuration variables that can be set within Nets. The Nets configuration in Site.pm can be overridden by a range of per-user and per-host sources, such as configuration files, command-line arguments, and User preferences in the database. Changing the configuration variables allows you to configure the default behaviour of Nets. Changing them in Site.pm makes those changes apply to all users who run Nets on the hosts where your Site.pm is installed, but still allows them to be overridden on a per-user or per-site basis.

Site.pm is amongst the last of the Nets support modules loaded when Nets starts up. It can therefore be used to override most functions and data within Nets.

You can also add site-specific code to Site.pm, and use that code to establish Hooks, change the user interface definitions or schema or change or override any code that is part of Nets. However, unless special conditions exist, we recommend that site-specific code be put in a Nets Plug-in (see Section 7.0 on page 17).

6.0 Hooks

Hooks are a Nets mechanism which allow your own Perl code to get control at specific times during normal Nets operation. For example, you can register a hook that will run your own code every time a Nets object is deleted from the database, perhaps to add or remove some site specific data from an external file.

Nets implements a number of hooks, each serving a specific purpose. Each hook has a name, and is run at a specific time during Nets processing. You can register any number of callbacks for the same hook; they will be run in the order in which they were registered.

Here is some typical code to create and register a hook:

```
sub logSomething
{
    my ($p, $s, $self) = @_;
    print "here I am in the registered logger: $p, $s\n";
}
.....
&Nets::Hooks::register('log', [\&logSomething, $self]);
```

When the hook is subsequently run, the caller will pass a fixed number of arguments (`$p` and `$s` in the example above). The arguments depend on which hook is being called, but usually provide some information about what is being affected or operated on at the time the hook is run.

When the hook is registered, the registering function can also specify some private data to be passed to the hook each time it is called (`$self` in the example above).

6.1 `obj_pre_load`

Called just before an object is loaded from the database:

```
hook($type, $where, $order, ....)
```

- `$type` is the name of the object type (and also the database table name)
- `$where` is the SQL where clause that will be used to select the required object (typically something like `ID=nnn`)
- `$order` is an optional SQL order by clause

6.2 `obj_post_load`

Called just after an object is loaded from the database and a new `Nets::Object` instance created to hold the contents:

```
hook($type, $where, $order, $self, ....)
```

- `$type` is the name of the object type (and also the database table name)
- `$where` is the SQL where clause that will be used to select the required object (typically something like `ID=nnn`)
- `$order` is an optional SQL order by clause

- \$self is a reference to the newly created Nets::Object

6.3 obj_pre_insert

Called just before a new object is inserted into the database:

```
hook($self, ....)
```

- \$self is a reference to the object about to be inserted. \$self->value('ID') will not be set until after the object has been inserted.

6.4 obj_post_insert

Called just after a new object is inserted into the database:

```
hook($self, ....)
```

- \$self is a reference to the object that has been inserted. \$self->value('ID') is set to the ID of the new object.

6.5 obj_pre_update

Called just before an existing Nets object is updated in the database:

```
hook($self, $oldobj, ....)
```

- \$self is a reference to the new version of the object
- \$oldobj is a reference to the state of the object the last time it was read from the database.

6.6 obj_post_update

Called just after an existing Nets object is updated in the database:

```
hook($self, $oldobj, ....)
```

- \$self is a reference to the new version of the object
- \$oldobj is a reference to the state of the object the last time it was read from the database.

6.7 obj_pre_delete

Called just before a Nets object is deleted from the database:

```
hook($self, ....)
```

- \$self is a reference to the object that is about to be deleted

6.8 obj_post_delete

Called just after a Nets object is deleted from the database:

```
hook($self, ....)
```

- \$self is a reference to the object that has just been deleted.

6.9 audit_insert

Called just before an event record is added to the Audit Trail (EVENT) table:

```
hook($etype, $otype, $oid, $desc, ....)
```

- \$etype is the event type. See `$Nets::Audit::ET_*`
- \$otype is the integer object type, which matches an entry in the OBJECTTYPE table
- \$oid is the ID of the object that was affected by the event
- \$desc is some descriptive text saying what the event was about

6.10 db_connect

Called just before a `Nets::DBSQL` connects to its SQL database:

```
hook($dbsource, $dbusername, $dbauth, ....)
```

- \$dbsource is the DBI database source string, typically something like: `'dbi:mysql:nets'`.
- \$dbusername is the username that will be used to log in to the nets SQL database
- \$dbauth is the password that will be used to log in to the nets SQL database

6.11 log

Called when a Nets log event occurs:

```
hook($p, $s, ....)
```

- \$p is the priority level of the log message, between `$main::LOG_ERR` and `$main::LOG_DEBUG`
- \$s is the message associated with the event

7.0 Plug-ins

Plug-ins are the primary and preferred method for inserting new functionality into Nets. A Plug-in is a Perl program module that is loaded into Nets when the Nets main program (`netsmain.pl`) start up.

Plug-ins can contain any Perl code, and have first-class access to all Nets data structures and functions, exactly like the core Nets code.

Using Plug-ins, you can, for example:

- Add new types of user information windows.
- Add new menu items to the Nets main window.
- Register a hook that gets control whenever a log event occurs and send it to another logging system, like `syslog`.
- Add site-specific business logic when certain types of objects are added or deleted from the database.
- Add new types of objects to the database schema and `ObjectEdit` user interface.

- Add new tools that do things to your network devices, such as upload a configuration file, etc.

Plug-ins (and other Nets files) can be distributed to other Nets users as part of a Nets Package (see Section 10.0 on page 22). You are encouraged to publish your Nets Plug-ins, or send them to us, so we can include them in the Nets goodies or the Nets core distribution.

When `netmain.pl` starts, it looks in the `<PluginsPath>` directory for files with the extension `.npm` (for Nets Plug-in Module). Any such files found will be loaded into Nets with the Perl 'do' operator. Perl compilation or loading errors will be reported and logged. Therefore it is only necessary to put a Plug-in file into the `<PluginsPath>` and it will be loaded next time `netmain.pl` starts.

The Plug-ins are loaded into `netmain.pl` after:

- the database is connected
- the current Nets user is logged in
- the Nets Main window is created and
- the configuration file and command line arguments have been loaded into the Nets configuration,

but before:

- any Nets scripts mentioned on the command line are started.

7.1 Deferred loading of Plug-ins

Often a Nets Plug-in requires many other files to be 'require'd or 'use'd in order to support its features. This can have an impact on the time required for Nets to load and start running. You can reduce the effect of slow loading of large Plug-ins by using deferred loading. Using deferred loading, the Plug-in includes a single function that loads the main part of the Plug-in only when it is needed. You can then put the main part of the Plug-in in a second file (without the `.npm` extension) and load it manually when your Plug-in function is called:

```
# mytestplugin.npm
$main::top->{file_menu}->command
    (-label => 'Just testing',
     -command => ~my_test_fn,
    );
sub my_test_fn
{
    require 'plugins/xxx.pm'; # Deferred loading
    return Nets::XXX->new(@_);
}
```

8.0 Database load files

Nets includes mechanisms for loading and importing data into its SQL database. A Nets Database Load File is a specially formatted data file that inserts and updates records in the Nets SQL database according to the data definition in Schema.pm (see Section 5.1 on page 7).

Database load files such as `basicdb.dat` in the Nets distribution are used to initially populate the Nets database at installation time. They are also used to create the objects as part of a Template (see Section 9.0 on page 21).

A Database load file can be imported into Nets in any of three ways:

- Using File->Import Database... in Nets.
- Using `netsload.pl`.
- Using Edit->Create from Template... in Nets.

8.1 File Format

A Load File is an ASCII text file that contains a sequence of multiline records. Here is a typical example:

```
# Nets database export Tue Mar  7 11:10:41 2000
Type:DEVICETYPE
ID:1
NAME:Cisco 3000

Type:DEVICE
ADDRESS:1
ASSET:A00001
COMMISSIONDATE:936280800
DESCRIPTION:Main router for Melbourne
DEVICETYPE:1
ID:1
INSTALLDATE:933602400
INTERNALTO:
LOCATION:1
MAINTAINER:
MANUFACTURER:
NAME:melcor1
NOTES:Leased from Cisco\nsecond line
OPERATINGSYSTEM:IOS 11.3
PURCHASEDATE:930924000
RACK:
SUPPLIER:
```

Any line beginning with a hash (`#`) is ignored.

The first line specifies the type of record that follows:

```
Type:DEVICE
```

The record type must be one of the tables defined by Schema.pm, otherwise the rest of the record will be silently ignored.

The record is terminated by a blank line.

Following the record type line, there is one line for each column value. Columns that are defined in Schema.pm, but which are not present in the record are set to NULL. Columns that are defined in the record, but which are not defined in Schema.pm are silently ignored.

Each column consists of a column name followed by the data for that column:

```
COLUMNNAME:here is the data
```

Any '\n' that appears in the data will be replaced by a newline character.

8.2 Unique IDs

Most tables in the Nets database are defined in Schema.pm with one column of type 'id'. Such ID columns are automatically maintained by Nets so as to ensure that column can serve as a unique identifier for each row in the table. When a database load file is loaded, Nets handles such ID columns specially.

If the load record does not specify a value for the ID column, Nets will automatically allocate a new unique ID for the new row.

If the load record does specify a value for the ID column, Nets will replace any existing row in the database of the same ID with the new record.

This mechanism means that you can choose whether to replace an existing record with the same ID, or to add the new record, regardless of what records are already in the database.

Using the `idForName` macro, you can also arrange to replace a record where you know the name, but not necessarily the ID:

```
Type:DEVICETYPE  
ID:idForName('Cisco 3000')  
NAME:Cisco 3000
```

This code will replace any existing row in the DEVICETYPE table that has a NAME column of 'Cisco 3000'.

8.3 Macros: Making relations between records

Nets load files permit a number of special macros for automatically setting IDs and foreign keys. These macros are intended to help in setting up relations between records in the SQL database. This allows you to ensure that when you create a DEVICE and its INTERFACES, for example, the DEVICE column in each INTERFACE refers to the ID

of the newly created DEVICE. That way you can ensure mutual consistency between all the records created by a load file.

The following macros are supported:

8.3.1 lastId('tablename')

Replaced by the ID of the last record created in the table 'tablename' by this load file. If there was no previous record for 'tablename' replaced by empty string. For example:

```
Type:DEVICE
DESCRIPTION:Cisco 3000
.....

Type:INTERFACE
DESCRIPTION:BRI interface for Cisco 3000
ID:lastId('DEVICE')
.....
```

8.3.2 idForName('name', 'tablename')

Replaced by the ID of the record in 'tablename' whose NAME column has the value of 'name'. If 'tablename' is not specified, it defaults to the table of the record currently being defined. This is especially useful for setting values for foreign key columns (defined as type 'fk' in Schema.pm). For example:

```
Type::DEVICETYPE
NAME:Cisco 3000

Type:DEVICE
DESCRIPTION:Cisco 3000
DEVICETYPE:idForName('Cisco 3000','DEVICETYPE')
```

9.0 Templates

Templates provide an easy way for Nets users to create a number of related objects in the Nets database. A Template is typically used for creating a particular type of DEVICE, along with all its INTERFACES and/or CONNECTORS.

Nets users are encouraged to contribute their Templates so they may be included in future releases for the benefit of other Nets users.

Most of the work of a Template is done by a Nets Database Load File (see Section 8.0 on page 19). But a Template also requires an entry in the Nets TEMPLATES table so that the user can find and select that Template.

The Load file specified by a TEMPLATE is required to exist in the <NetsDir>/templates directory. You can install a template load file in the <NetsDir>/templates directory and also create the TEMPLATES table entry by using the features provided by Packages (see Section 10.0 on page 22).

Although it is possible to automatically create a Template load file from a DEVICE using Nets, a Template that is intended for distribution to other Nets systems should be

hand crafted. This is because you need to be sure that any foreign keys are resolved to the correct row in the database.

That usually means that you *must* use `lastId` or `idForName` for any ID columns, and use `idForName` for all foreign key columns. If you don't do this, there is no guarantee that IDs and foreign keys will specify the correct records.

10.0 Packages

Packages are a way of combining one or more Nets modifications and extensions into a single easy to distribute and install bundle. A Package is the most common way for third parties to add significant new features to Nets.

A Package is a directory containing a number of files. It *must* contain a DESCRIPTION file. The DESCRIPTION file contains a description of the package, a list of the files to be installed and actions that are to be performed when the package is installed.

A Package can contain 0 or more of any of the following items:

- Files to be installed anywhere in the <NetsDir> directory (such as Plug-ins, Templates, Scripts, icons, drawings, maps etc.).
- Pre-Installation perl scripts
- Post-Installation perl scripts.

A Nets Package can be installed by one of two methods:

- File->Install a Nets Package in the Nets main program.
- `netspkginstall.pl`.

In either case, installation of a Package follows these steps:

1. The DESCRIPTION file in the package directory is located, read and parsed.
2. PreInstall scripts are loaded into Nets and executed in the order in which they appear in the DESCRIPTION file.
3. The files named by each NetsDir line in the DESCRIPTION file are installed into NetsDir.
4. PostInstall scripts are loaded into Nets and executed in the order in which they appear in the DESCRIPTION file.
5. An entry is added to the Nets PACKAGE table indicating what Package was installed, by whom and when.

10.1 DESCRIPTION File Format

Package DESCRIPTION files are ASCII text files that describe the Package, and specify what files are to be installed where, and what actions are to be taken at install time.

Line beginning with hash ('#') are ignored.

The file consists of a series of “keyword value” lines.

The following keywords are supported:

10.1.1 Name

Specifies the name of this package. This will appear on the Nets Package Installer window when the package is selected, and will also be inserted in the PACKAGE table when the package is installed.

10.1.2 Vendor

The name of the organization that created this package. This will be inserted in the PACKAGE table when the package is installed.

10.1.3 Revision

The revision number of this package This will be inserted in the PACKAGE table when the package is installed.

10.1.4 Description

A brief description of the package. This will be inserted in the PACKAGE table when the package is installed.

10.1.5 InstallBase

This optional keyword alters the source location of any files read or used during Package installation. The default is the same directory where the DESCRIPTION file was found.

10.1.6 RequireNetsVersion

This optional keyword specifies a minimum Nets version required to support this package. The package can not be installed into a Nets system with an earlier version number.

10.1.7 PreInstall

Each PreInstall line specifies a Perl file that will be loaded into Nets and executed before any files are installed. The PreInstall files will be run in the order that they appear in the DESCRIPTION file.

Because PreInstall files are loaded into Nets, they have full access to all Nets data and functions. Probably the most important data items are the Nets configuration variables from Site.pm (see Section 5.3 on page 14), and the SQL database access functions in DBSQL.pm (see Section 4.1 on page 5) and DBUtil.pm (see Section 4.2 on page 5).

10.1.8 NetsDir from[:to]

Specifies a file that is to be installed from the package directory (or InstallBase, if defined) into the current NetsDir. If the file named by `from` contains a leading path, it will be installed under the same path in NetsDir. For example:

```
NetsDir templates/tnt.dat
```

will install the file `tnt.dat` from the `templates` directory in the package into `<NetsDir>/templates`.

If the `:to` is specified, the file will be installed in the directory named by `to` in `NetsDir`. For example:

```
NetsDir xxx:templates
```

will install the file `xxx` from the package directory into `<NetsDir>/templates`.

Files will be installed in the order in which they appear in the `DESCRIPTION` file. Any previously existing files with the same name in the target directories will be replaced.

10.1.9 CgiBinDir from[:to]

Specifies a file that is to be installed from the package directory (or `InstallBase`, if defined) into the the directory defined by `CgiBinDir` in `Site.pm`. If the file contains a leading path, it will be installed under the same path in `CgiBinDir`. For example:

```
CgiBinDir private/myscript.pl
```

will install the file `myscript.pl` from the `private` directory in the package into `<CgiBinDir>/private`.

If the `:to` is specified, the file will be installed in the directory named by `to` in `CgiBinDir`. For example:

```
CgiBinDir myscript.pl:private
```

will install the file `myscript.pl` from the package directory into `<CgiBinDir>/private`.

Files will be installed in the order in which they appear in the `DESCRIPTION` file. Any previously existing files with the same name in the target directories will be replaced.

10.1.10 PostInstall

Each `PostInstall` line specifies a Perl file that will be loaded into Nets and executed after any files have been installed. The `PostInstall` files will be run in the order that they appear in the `DESCRIPTION` file.

Because `PostInstall` files are loaded into Nets, they have full access to all Nets data and functions. Probably the most important data items are the Nets configuration variables from `Site.pm` (see Section 5.3 on page 14), and the SQL database access functions in `DBSQL.pm` (see Section 4.1 on page 5) and `DBUtil.pm` (see Section 4.2 on page 5).

10.2 Bundling

A Nets Package consists of a directory containing a number of files, including at least a `DESCRIPTION` file.

It is common to want to send a Package to another Nets user, so that they can install the Package into their Nets system. The recommended way of bundling a package is in a gzipped tar file (also known as a *tarball* or *tgz* file). The bundle should be set up so that when the recipient untars it, it unpacks the package directory into the current directory. This can best be achieved by tarring the package from the next highest directory level. For example, imagine that you have made a Package that adds new network visualiza-

tion tools to Nets. The Package consists of a number of file in the directory 'visualiser' (including of course the DESCRIPTION file that says what is to be installed and where). If you are currently in the visualiser directory:

```
$ cd ..
$ tar cvf - visualiser|gzip -c >visualiser.tgz
visualiser/
visualiser/DESCRIPTION
visualiser/.....
.....
$
```

You can then send the visualiser.tgz file to anyone else to install into their Nets system. The recipient can then unpack and install the package with something like:

```
$ gunzip -c visualiser.tgz|tar xvf -
visualiser/
visualiser/DESCRIPTION
visualiser/.....
...
$ netspkginstall.pl visualiser
$
```

10.3 Examples

Here is an example that installs a new template file into the NetsDir templates file, and then adds a corresponding entry to the TEMPLATES table

- DESCRIPTION

```
# Example Nets package DESCRIPTION file
Name Example Package
Vendor Open System Consultants
Revision 1.1
Description Simple example of a Nets package

NetsDir templates/tnt.dat
PostInstall postinstall.pl
```

- postinstall.pl

```
# This is the postinstall perl script, which will be run
# after the files are installed. Adds an entry to TEMPLATES
# using a database load file
&Nets::Util::load
("$main::current_package->{InstallBase}/addtemplate.dat");
```

- addtemplate.dat

```
# addtemplate.dat
# Adds an entry to the TEMPLATES table
Type:TEMPLATE
DESCRIPTION:A sample template which makes an Ascend TNT
FILENAME:tnt.dat
NAME:TNT device
```

- tnt.dat

```
# This template creates a TNT device
Type:DEVICE
DESCRIPTION:Ascend TNT
.....
```

11.0 Scripts

Nets provides a simple scripting language for mechanizing some simple tasks. More complicated tasks are probably better suited to using a Plug-in (see Section 7.0 on page 17).

Scripts can be run using by either of two methods:

- File->Run Script ... in the Nets main program.
- As a command line argument to netsmain.pl.

Any number of Nets scripts can be running at the same time. Each script runs independently and in parallel with other Nets activities. Scripts are executed one line at a time whenever the Nets main program is otherwise idle.

11.1 File Format

Nets scripts are ASCII text files containing Nets Scripting Language (NSL) commands. The conventional filename extension is `.nsl`, but scripts can be given any name you wish.

The following NSL commands are supported:

11.1.1 print

Prints a string to stdout.

```
print here I am in this script
```

11.1.2 show

Creates or raises a Nets window. You can specify a particular type of Nets window to show, or let Nets choose an appropriate editing window for a certain type of Nets object. You can make the Nets window display a particular database object, and control its geometry (i.e. size and position) and title.

The general syntax for show is:

```
show [class Nets::windowclass] [type tablename] [id nnn] [title
sometitle] [geometry WxH+X+Y]
```

- *class* is the name of a Nets window class (e.g. Nets::Objectedit or Nets::report). The named window class will be loaded and instantiated. If not specified, *type* will be used to determine the preferred editor window class for the object type.

- *type* is the Nets object type to be display in the window. For some types of window (e.g. Nets::ObjectEdit) this will determine the exact type window which will be displayed.
- *id* specifies the unique ID of the object which is to be displayed in the window.
- *title* specifies the title which will be used for the window.
- *geometry* specifies the required width/height and/or position.

Show looks for an existing window with matching class and type. If one is found, that window will be reconfigured and raised. If no matching window is found, the appropriate Nets::class module will be loaded, and a new window created.

Hint: you can change the preferred editor class for a given object type by using Nets::Session::registerEditorClass.

11.1.3 list

Similar to show, except that if *class* is not specified, but *type* is, Nets will select the preferred lister class for that type of object.

Hint: you can change the preferred lister class for a given object type by using Nets::Session::registerListerClass.

11.1.4 run

Starts the execution of a Nets script. Does not wait for the new script to complete. The script named must be a file in the <NetsDir>/scripts directory.

```
run myscript.nsl
```

11.1.5 load

Loads a Nets Database Load file. The file name should be a full file path name.

```
load /usr/local/nets/files/sample.dat
```

11.1.6 message

Display a message string on a modal popup message box.

```
message This is a popup message box
```

11.1.7 install

Pops up a Nets Package Installer window, ready to install the package in the given directory.

```
install /usr/local/etc/nets/packages/visualiser
```

11.1.8 sleep

Suspends execution of the script (but not the rest of Nets) for the given number of seconds.

```
sleep 5
```

11.1.9 exit

Causes Nets to exit with an optional exit code.

exit 5

12.0 Getting Help

If you need help with Nets development tasks, you should consider using the Nets mailing list, or (if you have a support contract) the Nets support email address.

However, before posting to request information or help, try the following:

- Read and reread the manuals and guides, especially this Developers Guide.
- Examine the Nets source code, looking for example usage.
- Look for example code in the plugins or goodies directories.
- If you have a question or problem with Perl coding or syntax, you probably won't get much help from the Nets mailing list. Try a Perl book or an appropriate Perl mailing list.

If you solve your problem, consider posting a summary and the solution, so we can add it to the Nets FAQ.

If you find a bug, please let us know so we can fix it. The best way to let us know is to post to the Nets mailing list.