# Using the GNUstep AppKit

A guide to using various parts of the GNUstep AppKit for application development

**Christopher Armstrong**

# Table of Contents

# 1 Introduction

This manual documents some configuration and installation issues with the GNUstep GUI Library and also differences between the GUI Library and libraries that implement the OpenStep AppKit specification and the MacOS-X AppKit implementation.

## 1.1 Overview

The GNUstep GUI Library is a library of objects useful for writing graphical applications. For example, it includes classes for drawing and manipulating graphics objects on the screen: windows, menus, buttons, sliders, text fields, and events. There are also many peripheral classes that offer operating-system-independent interfaces to images, cursors, colors, fonts, pasteboards, printing. There are also workspace support classes such as data links, open/save panels, context-dependent help, spell checking.

It provides functionality that aims to implement the '`AppKit`' portion of the OpenStep standard. However the implementation has been written to take advantage of GNUstep enhancements wherever possible.

The GNUstep GUI Library is divided into a front and back-end. The front-end contains the majority of implementation, but leaves out the low-level drawing and event code. Different back-ends will make GNUstep available on various platforms. The default GNU back-end currently runs on top of the X Window System and uses only Xlib calls for graphics. Another backend uses a Display Postscript Server for graphics. Much work will be saved by this clean separation between front and back-end, because it allows different platforms to share the large amount of front-end code. Documentation for how the individual backends work is coverered in a separate document.

## 1.2 Implementation Details

Following are some implementation details of the GUI library. These will mostly be of interest to developers of the GUI library itself.

### 1.2.1 Drag and Drop

The drag types info for each view is kept in a global map table (protected by locks) and can be accessed by the backend library using the function -

```
NSArray *GSGetDragTypes(Nsview *aView);
```

Drag type information for each window (a union of the drag type info for all the views in the window) is maintained in the graphics context. The backend can get this information (as a counted set) using -

```
- (NSCountedSet*) _dragTypesForWindow: (int)winNum;
```

Whenever a DnD aware view is added to, or removed from a window, the type information for that view is added to/removed from the type information for the window, altering the counted set. If the alteration results in a change in the types for the window, the method making the change returns YES.

```
- (BOOL) _addDragTypes: (NSArray*)types toWindow: (int)winNum;
- (BOOL) _removeDragTypes: (NSArray*)types fromWindow: (int)winNum;
```

The backend library should therefore override these methods and call 'super' to handle the update. If the call to the super method returns YES, the backend should make any changes as appropriate (in the case of the xdnd protocol this means altering the XdndAware property of the X window).

You will notice that these methods use the integer window number rather than the NSWindow object - this is for the convenience of the backend library which should (eventually) use window numbers for everything

### 1.2.2 NSWorkspace

Here is (I think) the current state of the code (largely untested) -

The make_services tool examines all applications (anything with a .app, .debug, or .profile suffix) in the system, local, and user Apps Directories.

In addition to the cache of services information, it builds a cache of information about known applications (including information about file types they handle).

NSWorkspace reads the cache and uses it to determine which application to use to open a document and which icon to use to represent that document.

The NSWorkspace API has been extended to provide methods for finding/setting the preferred icon/application for a particular file type. NSWorkspace will use the 'best' icon/application available.

To determine the executable to launch, if there was an Info-gnustep.plist/Info.plist in the app wrapper and it had an NSExecutable field - use that name. Otherwise, try to use the name of the app - eg. foo.app/foo The executable is launched by NSTask, which handles the addition of machine/os/library path components as necessary.

To determine the icon for a file, use the value from the cache of icons for the file extension, or use an 'unknown' icon.

To determine the icon for a folder, if the folder has a '.app', '.debug' or '.profile' extension - examine the Info.plist file for an 'NSIcon' value and try to use that. If there is no value specified - try foo.app/foo.tiff' or 'foo.app/.dir.tiff'

If the folder was not an application wrapper, just try the .dir.tiff file.

If no icon was available, use a default folder icon or a special icon for the root directory.

The information about what file types an app can handle needs to be in the MacOS-X format in the Info-gnustep.plist/Info.plist for the app - see http://developer.apple.com/techpubs/macosxserver/System/Documentation/Developer/YellowBox/Re

In the NSTypes fields, I used NSIcon (the icon to use for the type) NSUnixExtensions (a list of file extensions corresponding to the type) and NSRole (what the app can do with documents of this type). In the AppList cache, I generate a dictionary, keyed by file extension, whose values are the dictionaries containing the NSTypes dictionaries of each of the apps that handle the extension.

I tested the code briefly with the FileViewer app, and it seemed to provide the icons as expected.

With this model the software doesn't need to monitor loads of different files, just register to recieve notifications when the defaults database changes, and check an appropriate default value. At present, there are four hidden files used by the software:

'`~/GNUstep/Services/.GNUstepAppList`'
: Cached information about applications and file extensions.

'`~/GNUstep/Services/.GNUstepExtPrefs`'
: User preferences for which apps/icons should be used for each file extension.

'`~/GNUstep/Services/.GNUstepServices`'
: Cache of services provides by apps and services daemons

'`~/GNUstep/Services/.GNUstepDisabled`'
: User settings to determine which services should not appear in the services menu.

Each of these is a serialized property list.

Almost forgot - Need to modify NSApplication to understand '-GSOpenFile ...' as an instruction to open the specified file on launching. Need to modify NSWorkspace to supply the appropriate arguments when launching a task rather than using the existing mechanism of using DO to

request that the app opens the file. When these changes are made, we can turn any program into a pseudo-GNUstep app by creating the appropriate app wrapper. An app wrapper then need only contain a shell-script that understands the -GSOpenFile argument and uses it to start the program - though provision of a GNUstep-info.plist and various icons would obviously make things prettier.

For instance - you could set up xv.app to contain a shellscript 'xv' that would start the real xv binary passing it a file to open if the -GSOpenFile argument was given. The Info-gnustep.plist file could look like this:

```
{
  NSExecutable = "xv";
  NSIcon = "xv.tiff";
  NSTypes = (
    {
      NSIcon = "tiff.tiff";
      NSUnixExtensions = ( tiff, tif );
    },
    {
      NSIcon = "xbm.tiff";
      NSUnixExtensions = ( xbm );
    }
  );
}
```

## 1.3 Contributing

Contributing code is not difficult. Here are some general guidelines:

- FSF must maintain the right to accept or reject potential contributions. Generally, the only reasons for rejecting contributions are cases where they duplicate existing or nearly-released code, contain unremovable specific machine dependencies, or are somehow incompatible with the rest of the library.
- Acceptance of contributions means that the code is accepted for adaptation into libgnustep-gui. FSF must reserve the right to make various editorial changes in code. Very often, this merely entails formatting, maintenance of various conventions, etc. Contributors are always given authorship credit and shown the final version for approval.
- Contributors must assign their copyright to FSF via a form sent out upon acceptance. Assigning copyright to FSF ensures that the code may be freely distributed.
- Assistance in providing documentation, test files, and debugging support is strongly encouraged.

Extensions, comments, and suggested modifications of existing libgnustep-gui features are also very welcome.

# 2 Basic Concepts

Throughout this manual, we refer to a number of concepts that you will need to be familiar with. It may be useful to at least glance over this section and make sure you are familiar with the concepts presented.

*application wrapper (or appwrapper)*

GNUstep applications rarely consist of just an executable file. They often contain interface files, property lists and other resources such as images. These are bundled together with the executable into a directory known as an *application wrapper*. To launch applications, you use the `openapp` command.

*delegate*

A *delegate* usually refers to an object that handles certain events and methods on behalf of another object. The methods a delegate should implement are declared as either a formal or informal protocol.

Many of the view and control classes within the AppKit allow you to supply delegate objects to help them make decisions about different things such as what data to display, how to handle events, whether to permit the user to select things, handling drag and drop, etc.

*formal protocol*

A *formal protocol* is a protocol that requires you to implement all the methods that are listed within it. They are used much less often that informal protocols in the AppKit.

Formal protocols are declared using their own statement, the `@protocol` identifier. You implement a formal protocol by placing it's name in arrow brackets ('<' and '>') and listing its methods in your interface declaration.

*informal protocol*

Objective-C can have both *formal* and *informal* protocols. Informal protocols don't require you to implement all of their methods.

Informal protocols are declared as a category of `NSObject`. You implement them in your own class by simply declaring and implementing the methods in the protocol you wish to. Always check the documentation of the classes that use the protocol to see which methods you should implement (usually at least one of them is mandatory).

*nib file*

*nib files* are your program's interface files, which contain definitions of all the windows, controls and menus as well as their connections to classes from your application. They have the extension '`.gorm`'.

See see [Interface Files], page 11 for more information.

*notification*

A *notification* is an indicator of a certain event. Notifications are objects posted to a *notification center*. Other objects may register with a notification center to receive notifications.

For more information about notifications, refer to the *GNUstep Base Programming Manual*.

# 3 GNUstep Applications

The AppKit provides services to develop complex and modern GUI applications. Some of these services include generic controls and displays, pasteboard and true drag-and-drop, separated interface and code files, etc.

Compared to other platforms and development toolkits, GNUstep takes a slightly different paradigm to development. Operating systems such as Microsoft Windows treat applications in a more window-centric manner, e.g. each document starts a new instance of the application, in a new window with its own menu.

In GNUstep, applications are treated in an application centric manner. This means that there is one menu for the application, and documents and other windows are associated with this menu instance. This probably requires a different attitude to development, but the AppKit is quite well integrated and logical to convey some of the ideas it introduces.

## 3.1 Components

A GNUstep application has various components that are assembled (from a developer's perspective) into an *app wrapper*. An app (application) wrapper is a directory with the extension '`.app`' that holds the application's *executable* and *resource files*, as noted below.

### 3.1.1 Interface File(s)

An application has one or more *interface files*. These are separate file entities that are used to display the graphical interface that your application has. They are comparable to .glade interface files used in GNOME or those used in Qt, however they go a bit further, permitting easy linking against your objects, so that you are freed from writing wrapper code. They are created using Gorm, GNUstep's application modelling programme. It allows real drag and drop GUI assembly and direct control editing.

Most applications take one interface file, which contains their main menu and their main window, presented to the user. They will also take you preferences and other auxilliary windows that you application requires. They take no Objective-C code (being strictly interface only), but generic class templates are able to be generated for outlets and actions that you set Gorm to integrate with.

Interface files are commonly referred to as "nib" files or "gorm" files, taken from the name of programmes used to generate them. They appear as a directory on your filesystem, and often take the name of your application with the extension '`.gorm`'.

### 3.1.2 Application property list

This file is a property list, containing the defaults and some information used to load your application, include the main interface file, supported document types and interapplication services. It usually takes the name '`Info-gnustep.plist`'. See the base manual for more details about the syntax and structure of property lists. We will provide the details of application property lists through this manual.

### 3.1.3 Makefile

Like GNUstep tools, applications have a file, '`GNUmakefile`', for easy application compilation, linking and assembly into an app wrapper. It includes the name and version of your application, source code file, required libraries and frameworks and your resource files (detailed below).

### 3.1.4 Resource Files

*Resource Files* are any sorts of resources that your application will need to operate, including interface files and any icons, images, data, etc. that your application uses. They are stored in the '`Resources`' directory in your application's app wrapper.

You will most likely ever need only two resources: your interface file, and your application's property list (Info-gnustep.plist).

## 3.2 Constructing an application

Below, we have listed the main steps required in the building of an application from scratch. These steps are listed in a general, but you will generally need to come back to them again e.g. if you add new source or interface files to your application, you will need to come back and modify the makefile. See the chapters on Makefile creation, Interface files and Application property lists for more details on the construction of these various files.

1. GNUmakefile

   You will need to create a GNUmakefile to build your application. A generic template is shown in the chapter entitled see [Application Makefiles], page 10.

2. Interface Files

   You will need at least one interface file ('`.gorm`') for your application, however, you can create your interface programatically if necessary (although this is rarely recommended).

3. Application Property List

   This is generally necessary, especially if you want to define your main interface file, however it is possible to let the `make` application generate it for you.

4. Other Resource Files

   These may include icons, images, other property lists, application-specific data files. You can add whatever resource files you like, including directories (which should be added recursively).

## 3.3 Building a First Application

This section attempts to run you through the steps that you would usually go through to assemble an application from scratch. We expect that you have some experience programming with Objective-C, especially with GNUstep, and that you at least have it installed and running with some applications installed.

Apart from helping you setup the infrastructure for a basic application, we've provided instructions for a basic control and event handler as an example. You may wish to ignore these steps, but they're useful reminders if you use these instructions in the future.

A checklist includes:

1. GNUstep Make sure that you have gnustep-make, gnustep-base, gnustep-gui and gnustep-back installed and running on your system. There system-specific instructions for installing GNUstep on different systems at the website, *http://www.gnustep.org*. We also expect that you have some experience using it, such as sourcing the GNUstep startup shell file and starting applications. There are various tutorials and instructions available on the internet for getting GNUstep up and running.
2. Gorm.app Gorm, as mentioned above, is the GNUstep interface builder. It's available as an application from the GNUstep web site, and is the recommended means to build interfaces for applications. Make sure that it will startup and operate correctly on your system. We will use it to build the interface for our application.
3. A text editor Depending on what platform your working on and whether or not you're using a GUI, an editor could be anything simple from `vim` to a good quality free editor like `gedit` or `kate`. You will need it to edit the source code files and makefiles we will use to build the application.
4. A shell GNUstep's makefile system depends heavily on the shell environment that `make` commands are invoked in. On Unix, this could be '`sh`', '`bash`', '`ksh`', '`csh`' or whatever

you prefer to work with. On Windows, you will want to use MSYS which comes with a minimal Unix-like shell (a port of `bash`) which is sufficient for use with GNUstep. If you use the installable binary version of GNUstep for Windows, you should have a copy of MSYS installed.

We will assume somewhat that you know your way around your filesystem using it, and that you know most basic commands for creating files, starting programmes, manipulating directory structures, etc.

### 3.3.1 Setup

Startup your shell and source GNUstep.sh from your GNUstep installation (if it's not sourced by default). Create a directory for your application sources to be created in. For example:

```
> cd ~
> mkdir firstapp
> cd firstapp
> . /usr/lib/GNUstep/System/Library/Makefiles/GNUstep.sh
>
```

In the above, we simply created a new directory under our home directory called '`firstapp`', changed into it and sourced our GNUstep installation (which in this case is under '`/usr/lib`'[1]).

Next we will create our makefile. Using your favourite editor, create a file called '`GNUmakefile`' (the case is important). In this case we're using `vim`:

```
touch GNUmakefile
vim GNUmakefile
```

And in the makefile, add the following:

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = FirstApp

FirstApp_OBJC FILES =  main.m  \
  MyController.m

FirstApp_MAIN_MODEL_FILE = FirstApp.gorm

FirstApp_RESOURCE_FILES = FirstApp.gorm

include $(GNUSTEP_MAKEFILES)/application.make
```

The first line of the makefile includes some declarations common to all makefiles (tools, bundles, etc).

*APP_NAME* contains a space-separated list of the applications to build. In this case we're only building one (FirstApp). The application wrapper that is outputted will be called '`FirstApp.app`'. This name also is used to prefix each of the following variables. If you were to change this value, you would have to change the value of *_OBJC_FILES*, *_MAIN_MODEL_FILE*, etc. accordingly.

*FirstApp_OBJC_FILES* contains a list of each of the Objective-C files that will be compiled into this programme. Note that like normal makefiles, you can split a variable declaration like this over a number of lines by using the "\" delimeter.

*FirstApp_MAIN_MODEL_FILE* is the main interface file wrapper.

*FirstApp_RESOURCE_FILES* contains a list of all the resources, including interface files, icons, property lists etc.

The final line lets the makefile system know we want to build an application.

[1] Replace '`/usr/lib`' with the path to your GNUstep installation

### 3.3.2 Interface File

Make sure you are familiar with Gorm before using this. Refer to the *Gorm Manual*, a link to which should be at http://www.gnustep.org/experience/Gorm.html.

Load up `Gorm.app` and create a "New Application". A window should appear with the title and a project called "UNTITLED".

```
openapp Gorm.app

From the menu, select Document->New Application
```

Save your project as "FirstApp.gorm" by going to Document->Save, navigating to the project directory, typing in "FirstApp" as the filename and clicking "OK".

Select your window in the project pane. Switch to the Inspector and give it a new name such as "My First Application".

Switch back to the project pane. Select "Classes" from the toolbar. From the class view, select "NSObject", goto the main menu and select Classes->Create Subclass.

Double-click the new class in the class view, and double-click to rename it to "MyController" (case is important). Click "OK" if prompted. We're going to use this class as our application's main controller, but you can create as many "controller" classes as you like with whatever names you choose. It just so happens that we've decided to create a file with the name "MyController.m" that will contain the implementation of this class. Note that GNUstep doesn't enforce a strict MVC pattern on your classes; it merely separates the view part into it's own classes which you configure in Gorm.app, and lets you handle data and behaviour (Model and Controller) in your code as you like.

Select the button in the "Action" column for "MyController" then goto Classes->Add Outlet/Action. Rename the action to "myAction:". Select the class again, and goto Classes->Instantiate. Again, we could call this action whatever we like, just make sure that it's not something generic like "click:", which are used by the NSResponder class. The name of the button in method name form is often a good choice.

For the MyController class, goto the main menu and select Classes->Create Class Files. Save them as "MyController.h" and "MyController.m". Gorm.app fills out the basic details for this class (including the action). If you modify the actions and/or outlets on the class in Gorm.app in the future, you will want to add them to your class interface and implementation manually. Gorm.app will override your modifications to files if you tell it to create the class files at some time in the future.[2]

Goto the palette, click the third toolbar button and then click and drag a new button object onto the window. Double-click the button to rename it and call it "My Action".

We now want to connect the button to the action on *MyController*. First switch to the "Objects" pane in the project view. Note that our *MyController* class is listed as an object instance, as we instantiated it before. Select it, switch to the Inspector and then select "Connections" from the drop-down box.

Now, make sure that the application window with the button on it and the project window are both visible at the same time. Hold down your first control key (usually left-Ctrl), click the button on the window, and drag the icon to the *MyController* object in the Objects pane and release. While you are dragging the mouse, you will note that the icon looks like a small circle with a "T" in it. The source object (the button) will continue to contain the "S" circle while the target object (the *MyController* instance) contains the "T" circle.

Goto File->Save to save your interface file and then quit Gorm.app.

---

[2] If you really don't want to add the outlets/actions to your class by hand if you modify the interface in the future, you could just save the classes to differently named files and then merge the changes back into the original files.

### 3.3.3 Code

Although we have got Gorm.app to autogenerate our class files, we will want to modify them so that they do something (marginally) useful. Open "MyController.m". At the moment it should look something like:

```
#import "MyController.h"
#import <AppKit/AppKit.h>

@implementation MyController

- (void) myAction:(id) sender
{
}

@end
```

# 4 Application Makefiles

Application makefiles are very similiar to those used to build Tools and Objective-C programmes, but allow extra specifications to build application wrappers and include their resource files. We assume you are already familiar with the GNUstep Makefile system.

Below is a generic, but complete application makefile, followed by an explanation of the various parameters.

```
include $(GNUSTEP_MAKEFILES)/common.make

APP_NAME = ExampleApplication
PACKAGE_NAME = ExampleApplication
VERSION = 1.0

ExampleApplication_OBJC_FILES = main.m AppController.m \
  ExampleClass.m

ExampleApplication_C_FILES = regexp.c fun.c

ExampleApplication_OBJC_LIBS = -lLibNumberOne -lPDFKit -lFunKit
ExampleApplication_RESOURCE_FILES = \
  ExampleApplication.gorm \
  Info-gnustep.plist

-include GNUmakefile.preamble
include $(GNUSTEP_MAKEFILES)/application.make
-include GNUmakefile.postamble
```

'`common.make`' and '`application.make`' are necessary to build an application, and need to be at the beginning and end respectively to the Makefile to operate properly. The '`GNUmakefile.preamble`' and '`GNUmakefile.postamble`' are optional, and permit you to define extra rules for building your application. You can include those lines without those files containing anything. Templates for those files also exist with the source code for gnustep-gui, which can simply be copied into your project and modified accordingly.

The table below describes the makefile variables that you can set to control the output of the make process. Note that *appname* refers to the application name that you set with `APP_NAME`. It is case sensistive and so are file names. Also, most of the variables listed below are optional if you wish to get a program to compile, but it is recommend you make use of them where appropriate. Where variables ask for flags and compiler options, they should be in the format that `gcc` expects, as it is the only compiler currently used with GNUstep. Many variables also take more than one parameter. They are usually separated by a space, and line breaks with a backslash. Please refer to the *GNUstep Makefile Manual* for more details.

`APP_NAME` [Required] This is the name of your application, and will be used to generate the name of your application wrapper.

`PACKAGE_NAME`
This is used to generate a rpm or deb package for distribution of your application. See the *GNUstep Makefile Manual* for more details.

`VERSION` A version number for your application.

*appname*_OBJC_FILES
: [Required] Replacing *appname* with the name of your application, you list the Objective-C files (.m), separated by a space. As shown above, you can split it across one or more lines by placing a slash at the end of the line to indicate a split.

*appname*_APPLICATION_ICON
: [Optional] You can place the name of the image file that will be used as your application icon here.

*appname*_MAIN_MODEL_FILE
: [Recommended] Put the name of your interface file ('`.gorm`') here. It will then be placed in the property list of your application.

*appname*_PRINCIPAL_CLASS
: [Optional] If you subclass `NSApplication` with your own application class, you should place it's name here. By default, GNUstep uses `NSApplication` as the application class.

*appname*_C_FILES
: [Optional] This is where you list the C source code files (.c) to be compiled into your programme. It takes the same form as *appname*_OBJC_FILES.

*appname*_CC_FILES
: [Optional] This is where you list your C++ files (*.cpp, *.cc) to be compiled into your programme. It takes the same form as *appname*_OBJC_FILES.

*appname*_OBJCC_FILES
: [Optional] This is where you list your Objective-C++ files (*.mm) to be compiled into your programme. It takes the same form as the *appname*_OBJC_FILES.[1]

*appname*_RESOURCE_FILES
: [Recommended] Here you list the *resource files* that are to be included with your application, including your application property list, interface file(s) and other images, data, etc. You can also list directories here, which should be added recursively (e.g. '`.gorm`' files are actually a directory containing three files, used to describe your interface).

*appname*_RESOURCE_DIRS
: [Optional] Here you can list directories that will be copied into your application wrapper as resources.

*appname*_OBJC_LIBS
: Here you list the names of the libraries you need your application to link against. Each one is prefixed by '-l' e.g. `-lMyLib`, separated by a space. You do not need to list the gnustep-gui, gnustep-base and Objective-C runtime, as these are included for you.

*appname*_C_FLAGS
*appname*_CC_FLAGS
*appname*_OBJC_FLAGS
*appname*_OBJCC_FLAGS
: Here you specify the flags to be passed to the compiler when processing this file type. These included warning flags and macro overrides.

[1] You will need gcc 4.1 or higher to compile Objective-C++ programmes. This feature of the gcc compiler is quite new and has not been well tested.

# 5 Interface Files

*Interface files* are used to store your applications graphical user interface. It's separation means that you can modify it more easily than other interface generation mechanisms, such as code generators, which generate code that makes it hard to modify the interface or requires you to rewrite your own code. The advantage of a separate interface file is that you can change the interface without having to recompile one bit of code (in most instances).

Interface files are often referred to as "nib" files.[1] These are not the same as those appearing on NeXT and MacOS X systems, and are completely incompatible (tagged XML nib's may change this in future). This section is very important to understanding key concepts in the AppKit, especially with regards to manipulation of your user interface. It is strongly recommended you do not skip this section, and refer back to it if you do not understand anything, even if you don't intend to use Gorm (also not recommended).

## 5.1 Using Gorm

Gorm is the GNUstep application used to create interface files for applications. It uses a drag and drop interface to place the control's on your window form and menu. See the Gorm manual, currently posted on the *GNUstep Wiki* for further information on using Gorm for the creation of interfaces. This section is also relevant to those using Renaissance.

### 5.1.1 The Palette

The palette contains pictures of various objects that you can drag and drop onto a window or a menu, including a window itself. These are the graphical objects that you may put onto your interface. They can be resized on the window itself by using the resize handles.

The graphical elements you place on your window(s) using Gorm, including the window itself, come from the palette. When your nib file is loaded, all the graphical elements are instantiated by GNUstep and all connections are made (see outlets and actions below). You don't need to instantiate objects in code, unless you intend to draw them programatically and add them to your interface. This differs from many other toolkits, where you often need to make connections to your interface in code (e.g. Win32 resource files) as well as instantiate custom objects for them, e.g. in Gtk, you need to add object variables that refer to the objects in your interface such as windows and buttons.

In GNUstep, you need only draw your interface and make connections to objects using Gorm, and then provide reference variables in the classes you specify connections (outlets and actions) for.

### 5.1.2 The Inspector

The inspector contains four sections, which let you modify the properties of any object (including those appearing on your window, the windows themselves and objects in the Objects Pane) in four sections:

`Attributes`
: This contains the attributes of the object you are modifying e.g. it's title, it's colour, tag items, etc. Note that fonts are modified using the Font Panel, which is opened in the menus, separately.

`Connections`
: Connections has three panes: the outlets, actions and object connections that you have made for this object (see see [Outlets and Actions], page 13).

[1] This is a throwback to the origin's of the GNUstep framework, where it's API specification (OpenStep) was based on NeXTStep, which used "nib" files (NeXT Inteface Builder) to store interfaces.

`Size` Lets you modify the size using numbers, and the resizing data for this object (the springs at the bottom). See the Gorm manual for more details on this pane.

`Help` Help related to this object type. Still being completed.

`Custom class`
Let's you set the class for this object manually (NOTE: this only appears for some objects where it is possible to set a custom class).

### 5.1.3 The Objects Pane

This is a graphical display of your application's objects. They appear with a subtitled icon, and can be manipulated like the graphical objects on the window forms, as well as be connected to outlets and actions.

You can also instantiate subclasses (where necessary) to connect to other objects. The object's here may be representative (for example, NSOwner and NSFirst) or be instances of custom classes you create in your code.

### 5.1.4 The Classes Pane

This permits you to subclass and add actions and outlets to classes that you want Gorm to link up at runtime. The GNUstep class hierachy is shown in this pane, allowing you to see the various views and helper classes (e.g. NSDocument/NSDocumentController, used for document based applications).

In here, you can create subclasses of the classes specified, often `NSObject` and then add actions (methods) or outlets (instance variables) to them. What you do in here must be reflected in your own code at the time your nib file is loaded. As a result, Gorm can generate the appropriate header and source files for you, or you can create them yourself. However you do this, you must make sure any subclasses you create here can be found at runtime, and that they contain all the instance variables and methods that you specify as outlets and actions at the time that your code loads the nib file (often by calling `-loadNibNamed:` on the main bundle).

### 5.1.5 Outlets and Actions: Connecting objects

Gorm permits you to connect your graphical objects together using it's interface, to save you the trouble of connecting them at runtime using extra lines of code (and wondering where you should put them). We introduce two concepts here: *outlets* and *actions*. They form the basis of event handling and graphical object linkage when using Interface Files for your programme's interface in GNUstep. This outlet and action paradigm is incredibly important, as you will see, as it eliminates the need for subclassing objects of graphical elements (a.k.a widgets) for all but the most complex GUI applications. You don't even need to subclass NSWindow to create a window; you merely need to instantiate it, and that bit is taken care of by GNUstep anyway (unlike most other GUI toolkits, including the Win32 API, Gtk, Qt/KDE, and many others).

The concept of *outlets and actions* is presented in many beginner tutorials to GNUstep and Cocoa. It is well recommended you follow one of these to get a better idea of how these things work. With practice, they become second nature in interface design, and are useful for thinking about how your interface will interact with your code, but still keeping a useful abstract distance between the two.

An *outlet* is a property of an object, that can be used to store a reference to another object, which is usually some sort of graphical element (like a button or text box). You usually add outlets to your custom subclasses and then connect them to graphical elements on your window, so that you can directly manipulate them in your code. If you were to add an outlet to a class, it would appear in code under the data value declarations part of your class as an object reference. It takes the syntax:

```
id myOutlet;
```

(NOTE: `id` may also be `IBOutlet`, especially if generated by ProjectCenter. It seems not to matter.)

For example, if you connect a button to the outlet of one of your objects, say and outlet called `myButton`, when that nib is instantiated, `myButton` will contain a reference to an `NSButton` object, namely the button object on your interface that is connected to that outlet.

Another example is creating a main window for your user interface. You may decide later that you wish to customise this window in code, based on user interactions. It would be appropriate to add an outlet to a top level object so that you can access this instance of the window.

You will often create a special subclass of `NSObject` named something like *AppController* or `ApplicationController` and instantiate it. You will then add outlets and actions to this so that you can centralise access to your programme's widgets. The default Application project type in ProjectCenter does this for you, and many tutorials will present outlets and actions to you like this.

An *action* is a method or function of behaviour that a class may perform. For example, you can connect a button to an action listed in NSOwner, so that when the button is clicked, it will send a message to the NSOwner object that will perform the action you connected. Actions are listed as methods on objects, but they take the form:

```
- (void) myAction:(id)sender;
```

Hence they are instance methods, taking one parameter, which is a reference to the object that is connected to the action. You could connect any number of objects to action on one object, so that it could distinguish between it's caller's by checking the sender object with GNUstep's introspection/reflection features.

For example, say that you create an action on one of your custom objects called `compute:`. If you then connect a button object to your custom object and set `compute:` as the action, when the button is clicked, it will call `compute:` on your custom object.

In short, objects are connected to outlets or actions, but outlets or actions are not connected to each other. To connect an object to an outlet or an action, you first select the object, then hold down the first control key (usually the left CTRL key on your keyboard), and select (using the mouse) the object which contains the outlet or action you wish to connect to. In the Inspector window, you select the target outlet or action, and click `Connect`. The action or outlet on the latter object will be connected to the first object.

### 5.1.6 NSOwner: Who controls your interface?

*NSOwner* will appear as an object in the Objects Pane. You will notice that the only property you can set is it's class. NSOwner is an object, decided upon at runtime by your code, that will "own" this instance of your interface. You can instantiate interfaces more than once upon runtime, each time associating an instance with a different object.

You can set NSOwner to be a custom class, with your outlets and actions, and then connect NSOwner to other objects or graphical elements or methods in your interface.

For example, you may create a custom subclass of NSObject called `MyController`. You may then give it a number of outlets, including one called `window`. You could set NSOwner to be of your subclass type, then connect `window` to the NSWindow object in the Object's pane. Upon runtime, whatever `MyController` object you set as NSOwner would have the associated NSWindow instance appear in it's `window` instance data value.

### 5.1.7 NSFirst

NSFirst is an abstract object, and may refer to any number of different graphical elements during the lifetime of your programme. It is what's known as the *first responder*, the object that is

connected in such a way to receive event's first. The first responder may change depending on user interaction with a window, e.g. selecting an object in your window may cause it to become the first responder.

What you can do is connect outlets and actions to the first responder, and depending on whether the object that is set as `NSFirst`, the use may be able to perform that action.

For example, you may connect a menu item to call the `print:` action on the NSFirst object. GNUstep will automatically grey out this menu item if the object set as the first responder (decided by the user's currently selected object) is not able to respond to this action. If another object is to be later set as the first responder, and is able to respond to this action, GNUstep will automatically make the menu item available again. This way, you don't have to handle instances where the first responder object cannot respond to your method. GNUstep sets this all up using Objective-C's introspection features, by checking whether your object responds to the method corresponding to the action.

## 5.2 Loading and Instantiating Interface Files

Once you've created your interfaces files, you will want to instantiate them and display them. This is relatively simple in code, and merely requires you deal with the *NSNib* class. If your application contains only one interface file, it is possible to avoid this step altogether, and set the main interface nib as a property in your application's property list, as well as including it as a resource in your application's makefile.

Otherwise, if you would like to instantiate it manually, especially if you have multiple interface files, you first create an NSNib object using the name of your interface file, and then instantiate it with a reference to your NSOwner object.

```
id myOwner;
NSNib* myNib;
NSArray* topLevelObjects;

// Assign myOwner to an object of the class that you set as NSOwner.

myNib = [[NSNib alloc] initWithNibNamed:@"MyNibFile" bundle:nil];
[myNib instantiateNibWithOwner:myOwner topLevelObjects:&topLevelObjects];
```

In this case, we first create the `NSNib` object `myNib` with a interface file called '`MyNibFile`'. We pass `nil` to the bundle parameter to indicate the main bundle; you can otherwise specify another bundle you may have already loaded into your programme. The `topLevelObjects:` parameter refers to the objects that appear in the Objects pane in Gorm. You can use Objective-C's reflection features to identify them.

The nib is then instantiated with myOwner as the NSOwner object (you have to create this beforehand) and is passed a pointer to an `NSArray` reference so that you can receive the top level objects in your interface file. See the NSNib documenation in the AppKit reference manual for more details. There is simpler methods calls available for instantiating nib's as well under `NSBundle` (see the *GNUstep GUI Reference Manual*.

# 6 Basic Controls

One of the first important concepts you will encounter dealing with the widgets in the AppKit is that of a *control*. A *control* is just a simple graphical element that you put onto your window, such as a button, a text field or an image. It is a specialisation of a the concept of a view (which are a bit more abstract), and hence introduces its own terminology.

Controls can easily be spotted in the *GNUstep GUI Reference Manual* as they are derived from the abstract superclass `NSControl`. Every control has two classes, one derived from `NSControl`, the control, and one derived from `NSCell`, the cell. A control is responsible for it's corresponding cell, and usually contains only one cell (although matrices and tables contain groups of cells).

The control hosts an instance of an NSCell subclass. This specific NSCell subclass can be set for a particular type of control by calling it's `+setCellClass` method, which will cause that NSControl to use your subclass instead of it's own for creating it's cell.

One can set the *value* of a control either directly or indirectly. You can directly set the value of a control by calling the `-setObjectValue:` method, or more specifically, the `-setStringValue:`, `-setIntValue:`, `-setFloatValue:` and `-setDoubleValue:` methods. We can also retrieve values using the `-objectValue`, `-stringValue`, `-intValue`, `-floatValue` and `-doubleValue` methods.

More indirectly, the control can be instructed to take it's value from another control when that control changes. We, the *receiver*, can take our value from another object, the *sender*, when the sender is updated. You can set what sender the receiver will take it's value from by calling the `-take*ValueFrom:` methods on the receiver, passing in a reference to the sender object. This mechanism only permits one-to-one relationships.[1]

The control can be enabled/disabled from receiving mouse events (as well as others) by setting the enabled property (`-setEnabled:`). You can tell the control to resize to the minimum needed to comfortably display it's cell by calling the `-sizeToFit` method.

With regards to the generation of actions, you can set the selector that the control will call on the first responder with the `-setAction:` method. For more information with regards to what an "action" is in the context of event generation, see see [Outlets and Actions], page 13.

## 6.1 Basic NSControl Classes

Classes that leverage the paridgm and concepts provided by NSControl are detailed below. Note that some of the more complex subclasses have dedicated chapters, such as NSTableView, NSTextView and NSMatrix.

### 6.1.1 Buttons (NSButton)

A button can be more than a simple "push button". This NSControl is used to implement radio buttons, momentary push buttons, radio style buttons, etc. The way the button reacts is specified by the `-setButtonType:` method, which takes a constant value of one of the following:

`NSMomentaryPushInButton`
`NSMomentaryPushButton`
: This is the default button type. It is "pushed in" and lit while the mouse is held down on it, and it is "pushed out" and unlit when the button is released. It is used for triggering actions; it doesn't graphically nor internally store an on/off state. It looks like a simple click button that you would find in Microsoft Windows.

`NSMomentaryLightButton`
`NSMomentaryLight`
: This type of button simply appears "lit" while the mouse is held down on it. Like the NSMomentaryPushInButton type, it used for simply triggering actions.

[1] You would model this in UML using a one-to-one association I think.

`NSPushOnPushOffButton`
: This button is used where you need to show and store an "on/off" state. When the button is first clicked, it is highlight and "pushed in" while the mouse is held down. It maintains this state until it is clicked again, in which it returns to normal.

`NSOnOffButton`
: This is like the NSPushOnPushOffButton, but it only highlights the button's area when clicked on and off.

`NSToggleButton`
: This type is an on/off button like NSPushOnPushOffButton. When it is clicked, it changes it's image to indicate an "on" state. A second click will restore the original button state.

`NSSwitchButton`
: The same as NSToggleButton, but with no border.

`NSRadioButton`
: A variation of NSSwitchButton that is similiar to the radio button control in Microsoft Windows.

A button has a "title" property, which is the text either displayed on or next to the button (depending on whether it's of the switch or push variety). This is changed with the `-setTitle:` method. The button state, as discussed above, can be read or changed with the `-state` and `-setState:` methods.

You can also set an image to be displayed on the button (`-setImage:`) as well as an *alternate image*, which is displayed when the button changes state (`-setAlternateImage:`). Along these lines, the button also has an alternate title which appears when the button changes into it's "on" state (set using the `-setAlternateTitle:` method). Both the title and alternate title can be set using attributed strings as well.

Another visual feature that can also be set is whether it is bordered (`-setBordered:`) and if so, what type of bezel that border takes (`-setBezelStyle:`).

### 6.1.2 Text Field (NSTextField)

A *text field* is a simple control that displays and/or allows the editing of text. You can set whether it is editable or not using the `-setEditable:` method.

It also can take a delegate implementing the `NSControlDelegate` protocol, which is described below.

### 6.1.3 Combo Boxes (NSComboBox)

A *combo box* is similar to a text box, but it also has a drop-down component that lets the user select from some predefined entries as well as letting them type one it.

You can provide the data it uses by calling methods on the object or setting a data source. Objects can be added to the list using `-addItemWithObjectValue:` or `-addItemWithObjectValues:` (for arrays), and then removed with `-removeItemWithObjectValue:` or `-removeAllItems`. The items listed can be referenced by index if necessary.

If you wish to use a data source, you must first set a data source object that implements the `NSComboBoxDataSource` informal protocol, and then call `-setUsesDataSource:` with a `YES` parameter.

### 6.1.4 ImageViews (NSImageView)

An *image view*, which displays an image, is also a control. You can set the image to be used with the `-setImage:` method. It is also possible to set the alignment, frame style and image scaling.

See see [Images and Imageviews], page 47 for more information.

### 6.1.5 Popup Buttons (NSPopupButton)

A *popup button* is a special kind of button that displays a menu while the mouse button is clicked and held down on it. The user selects an item from the menu by moving the cursor over the item they want and releasing the mouse button.

It can behave as a pull-down or a pop-up menu. You can change this using the `-setPullsDown:` method and providing a boolean. Items can be added and removed using `-addItemWithTitle:`/`-addItemsWithTitles:`, `-insertItemWithTitle:atIndex:` and `-removeItemWithTitle:`/`-removeAllItems`/`-removeItemAtIndex:` methods.

The selected item is retrieved via the `-selectedItem` method (and others). It posts one notification: `NSPopUpButtonWillPopUpNotification`, which is posted just before the menu is shown.

### 6.1.6 Scroller (NSScroller)

*Scrollers* are scrollbars. You will be unlikely to instantiate these directly, as scrolling functions are handled best by `NSScrollView`. Otherwise, their visual appearance and behaviour is very customisable.

You can otherwise get where the scroller is positioned by calling `-floatValue` which is a number between 0.0 and 1.0 (0 being at the top/left end and 1 at the bottom/right end). Similarly, the position and proportion of the knob that fills the knob slot can be set using the `-setFloatValue:knobProportion:` method (the proportion also being between 0.0 and 1.0).

### 6.1.7 Slider (NSSlider)

A *slider* looks alot like a scroller, but is simply a knob used to allow the user to select a variable value. If you want to allow the user to select a variable value, use this instead of a scroller.

It's value is set and retrieved via the `-setFloatValue:` and `-floatValue` methods defined in `NSControl`. It also permits a minimum and maximum value to be set.

You can set an image to be displayed in the scroll bar part using `-setImage:`, and you can set a title (and/or title cell/font/colour) to be shown with the slider.

When the user clicks and drags the slider, it will continually send it's action message as the user drags the slider. This behaviour can be changed using the `-setContinuous:` method.

### 6.1.8 Steppers (NSStepper)

A *stepper* is a control that displays it's current value in a box while permitting the user to change it via a pair of up/down arrows.[2]

Like the slider, you can set a maximum and minimum value. You can also set whether the value wraps, and by how much it is incremented/decremented on each mouse-click.

## 6.2 Advanced control classes

GNUstep also provides more advanced control classes, notable tableviews, matrices and browsers. Many of these are documented in subsequent chapters.

[2] It's like the Microsoft Windows Spin control

A *matrix* is a grid containing cells. It does not matter what type of cells are put into it, and they can be of different types, as long as they're all the same size (see [Matrix Controls], page 33). They are referenced by a cell coordinate number, and data is added passively via calling methods on the NSMatrix object.

*Tableviews* are different from matrices, essentially displaying grid lines and drawing column headers. They are more useful for displaying records of data from database tables and queries, amongst other things. They are organised by column (fields) and rows (records). Unlike matrices, they use a data source delegate to display their data. For more information, see [Tableviews], page 30.

*Browsers* are a useful control for displaying hierachial information, especially data that is subject to real time change or needs to be navigated in a hierachical fashion. They use a data source that can be either passive or active in the way it gives the browser data, so that you can have hierachies which change as the program runs, e.g. representing a file system (take a look at the GWorkspace program for a example of a browser control in use). For more information, see [Browsers], page 35

*Outline views* are a specialised form of table view that allows the display of hierachial data via rows that can be expanded and collapsed. They too use a special data source.

## 6.3 Control Notifications

Controls provide a few generic notifications, particularly related to text editing. All the following notifications will have the control that posted them as the notification object. The notification has a `userInfo` dictionary that has a key `@"NSFieldEditor"`, which is the editing cell's field editor.

`NSControlTextDidBeginEditingNotification`
: This notification is sent when a control has begun editing. This only applies to controls that are editable.

`NSControlTextDidEndEditingNotification`
: The notification is sent when a control has finished editing. This only applies to controls that are editable.

`NSControlTextDidChangeNotification`
: This notification is sent when the text in a control has changed. This only applies to controls that are editable.

## 6.4 Control Delegate

You can also set a control delegate by calling `-setDelegate:` on the control subclass with an object that implements the informal protocol `NSControlDelegate`.

The delegate receives the notifications defined above. If the control subclass has it's own delegate protocol(s), you may have to use the same object to implement both `NSControlDelegate` and the specific control's delegate.

## 6.5 Cell Classes

As previously mentioned, a controls' cell class inherits from `NSCell`. `NSCell` defines alot of basic functionality and features that cells can customise.

`NSCell` provides a number of methods for setting/getting the cell value. These correspond to those that are available for their corresponding control.

Like most graphical elements, cells can be enabled and disabled using the `-setEnabled:` method. They also have the concept of a `"state"` so that cells such as check boxes and radio buttons can

be defined as being "on" or "off". Cells may also have a "mixed" state, but this can only be enabled using the `-setAllowsMixedState:` method. The cell state can be retrieved using the `-state` method, which returns one of the foloowing constants:

`NSOnState`
: The cell is "on".

`NSOffState`
: The cell is "off".

`NSMixedState`
: The cell is in a "mixed" state. This may be, e.g. a checkbox representing a group of elements of which some are on and some are off.

In line with the target/action paridgm specified in previous chapters, cell's can have an action and a target set on them. The action is a selector, which can be retrieved using the `-action` method. The target is an object, which can be retrieved using the `-target` method. This stuff is usually setup by Gorm.app when you create your interface. You can set whether an action is continuous via the `-setContinuous:` method.

Cell's have a generic type. These can be retreived using the `-type` method and set using the `-setType:` method with one of the constants specified below:

`NSNullCellType`
: This cell doesn't display anything.

`NSTextCellType`
: This cell displays text.

`NSImageCellType`
: This cell displays an image.

The way cells display and format data or text can also be set. A formatter object that changes the way the cell's data is represented after the user has typed iit in is set via the `-setFormatter:` method using an object of `NSFormatter` derivation.

# 7 The view concept

This chapter discusses the concepts surrounding *views* and goes into some detail what can be done with them. As a result, most of this chapter is concerned with the creation of custom views, which is not necessary for general application development. If you want to create your own view classes or are interested in how GNUstep manages views, then this chapter should be useful.

## 7.1 Introduction

In GNUstep applications, we introduce the idea of a *view*. A view is a graphical element on the window in your interface. It is much like the idea of a window in the Microsoft Windows C API, except more powerful. Note that views are a *generalisation* of a control, that is, a control is a special type of view.

A view is a subclass of the AppKit `NSView` class. You should not instantiate this class directly, but instead use a class that is derived from it. A custom view can be created by inheriting from it.

## 7.2 The view hierachy

A view may contain any number of *subviews*, which are views that are displayed within it. Those views may also have subviews, and as a result, you can setup a hierachy of views. This can be a powerful model for your interface designs (especially where you create your views programatically instead of just in `Gorm.app`).

Each window has a primary view, known as the *content view*, which acts as a top-level view (or *superview*) to all the views you place on your window. It sits at the top of the view hierachy. Most applications will only have one level of views below the content view, and for most applications, this is all you need.

## 7.3 Frames and Bounds

As views can be placed within other views, GNUstep needs to setup some rules to determine how this will work. Due to this, each view has two important properties defining how it is positioned and displayed on the screen. These are it's *frame* and its *bounds*.

GNUstep uses cartesian coordinate systems for defining positions and sizes. It has the origin of any coordinate system placed at the bottom-left corner and has an x-axis and a y-axis. Like a normal cartesian coordinate system, the x-axis runs from left to right, and the y-axis runs from the bottom to the top. A view is defined within two coordinate systems, not just the coordinate system of the entire screen or window.

The frame and the bounds describe the view in terms of a rectanglei placed in a coordinate system. The rectangle has an origin (located at the bottom-left corner of the rectangle) and a width and a height. Programmatically, the concept of a rectangle is tied up in an `NSRect` structure, which in turn contains an `NSPoint` structure (for the origin) and an `NSSize` structure.

The contents of your view is not dissimilar to a canvas. You can draw anywhere on this canvas, but only a certain portion of it is displayed. Where it is displayed and what part of it you choose to display is defined in the frame and the bounds rectangle of the view.

The *frame* is the location and size of your view, as defined in its superview. The content view has it's frame defined with it's origin at the bottom-left corner of a window, and it's width and height equal to that of the window it is placed in (ignoring the window decoration). If you change the origin of your frame rectangle, you effectively move your view within it's superview. By changing the frame rectangle's width or height, you resize your view with regards to the coordinate system of it's superview.

The *bounds* rectangle defines what part of your view's internal coordinate system will be displayed. It is therefore defined in the coordinate system of your view. By default, it is set to be a rectangle located at the origin of your view's internal coordinate system, with it's size set to be the same size as your frame rectangle. However, it can be programatically streched, rotated, moved and skewed so that various parts of your view's internal coordinate system are displayed in it's frame rectangle.

In essence, the *frame* is defined by the coordinate system of your view's superview, and the *bounds* is defined by the coordinate system of your view. These concepts can be difficult to grasp, so we recommend you read over this bit, as well as play around with the various methods in GNUstep that let you modify the bounds and frame rectangles of a view.

It is the internal coordinate system where your view does it's drawing and which defines the location and size of any subviews. It is the coordinate system of your view's superview that defines where and how big your view is displayed.

## 7.4 Manipulating the coordinate system

As mentioned earlier, the bounds and frame of a view can be stretched, shrunken, shifted and even rotated. Firstly we will show you how to manipulate these directly, and then briefly describe the mathematics behind coordinate transformations.

NSView provides some simple methods for manipulating the coordinates and coordinate systems of the frame and bounds rectangles. Note that after calling any of these methods, you need to get the view to redisplay itself manually. You can do this by calling the `-setNeedsDisplay:` and `-display:` on the view o bject.

We can change where a view is placed within it's superview (most likely the window's content view) by manipulating it's frame origin. This is done using the `-setFrameOrigin:` method. For convenience, you can use the `NSMakePoint()` function to easily construct a point for the new location.

The size of a view's frame can also be adjusted using the `-setFrameSize:` method. Similiarly, the `NSMakeSize()` method can be used to construct an `NSSize` parameter that is needed. Changing this will cutoff whatever is internal to the view, although some classes behave differently. Check the documentation for the class with regards to it's reaction to a change in it's frame size.

Where necessary, these can be adjusted as a rectangle, making use of the `-setFrame:` method and the `NSMakeRect()` function.

Methods used for manipulating the bounds have subtly different meanings. Like the frame rectangle, the bounds rectangle can be manipulated as well.

The bounds origin and size can be manipulated using the `-setBoundsOrigin:` and `-setBoundsSize:` methods respectively. Changing the bounds origin effectively sets the new origin to be displayed at the origin of your frame rectangle. Changing the bounds size can be used to skew the coordinate system of the bounds, as it is displayed within the frame rectangle.

Another method for skewing the internal coordinate system of a view is to use the `-scaleUnitSquareToSize:` method. It's useful when you need to express your transformation as a percentage or fraction, where a size of 1.0 is considered to be 100%. Note that this method is cumulative, so that when you set this, it is effectively the first transformation multiplied by the second. For example, setting it to 0.5 and the 0.75 is the same as setting it to 0.5 x 0.75 = 0.375.

To rotate the frame or bounds rectangle counterclockwise, call the `-setFrameRotation:` or `-setBoundsRotation:` methods respectively. These methods take an angle in degrees. You can specify clockwise rotation with a negetive angle.

Alternatively, you can rotate the bounds rectangle by using the `-rotateByAngle:` method. This method rotates the bounds on top of what it has already been rotated.

## 7.5 Subclassing NSView

Sometimes the need arises to create a custom view. This is achieved by subclassing NSView. From here, you can override default event handlers and drawing methods to customise your view's representation.

Note that in some cases, the `NSControl` class may prove to be a better model for your custom view, especially if it behaves more like a control instead of an entire document representation. You should read the chapter on controls and weigh up the options for creating a view vs creating a control. This section is still useful though to understand the drawing code aspect, which is relevant to the display of control's as well.

### 7.5.1 Drawing code

One of the first things you will do is write your own drawing code. All custom drawing code is placed in the `NSView` method `drawRect:`. NSView's implementation is blank by default.

In this section, we will describe the various facilities at your disposal for drawing in a view.

### 7.5.2 NSBezierPath

This class is an abstract representation of a *bezier path*. A bezier path contains a series of straight and curved lines representations which come together to form a number of shapes that describe the object you wish to draw. This "path" can then be "filled", "stroked" or used as a clipping path within the view you are working on. It also contains the pen width, pen dash information and the current point.

A bezier path represents a series of graphical primitives operations. You call methods corresponding to these operations on an `NSBezierPath` object, and when it is drawn, these operations are executed in the order that you called them on the object.

It also has a concept of a *current point*. After every graphics operation, a new, internal point is set that will be used as the start point for the next operation. It is the destination point of the previous operation. For example, if you want a bezier path to move to the point origin, then draw a line from the origin to point (10,20), then a line from (10,20) to (20,20) you only require three operations, i.e. (in psuedocode):

```
move to (0,0)
draw a line to (10,20)
draw a line to (20,20)
```

In this case, the bezier path first sets the current point to (0,0). Then, when the line operation is called, you only pass in the destination point, (10,20), which causes it to draw a line from (0,0) to (10,20). After the line operation, the current point is set to the destination of the line operation, i.e. (10,20). Then, the next line operation draws a line from (10,20) to (20,20). In this way, we only need specify the destination point for line and move operations, as the start point is determined by the destination point of the previous operation. There is no need to specify the start point for each drawing operation, as it is implied by the destination point of the previous operation. You can get the current point by calling the `currentPoint:` method.

These operations are listed in the table below:

*Move Operation*
: A move operation lifts the pen up and puts it at a new location, i.e. changes the current coordinates without drawing. This can be achieved throught the `-moveToPoint:` method, which takes a point as it's first parameter. It implicitly begins a new sub-path (see below).

*Line Operation*
: A line operation draws a line from the current point to a new point. The *current point* is set either through a move operation, or through the last point in a previous

line or curve operation. Once the line operation is complete, the *current point* is set as the destination point. We can draw a line using the `lineToPoint:` method.

*Curve Operation*
: This one is more complex, as it involves the real magic of bezier paths. It consists of four points: the start point, the destination point, and two *control points*. How this works is beyond the scope of this manual[1] and is not required to draw simple circles, ellipses and arcs. We can draw a circle or an ellipse by calling `appendBezierPathWithOvalInRect:`, passing in a rectangle for the shape to be drawn in. A few methods are provided for adding arcs, with `appendBezierPathWithArcFromPoint: toPoint: radius:` useful for adding an arc between two points and the `appendBezierPathWithArcWithCenter: radius: startAngle: endAngle:` method useful for drawing an arc with a particular centre point. For those familiar with bezier curves or who know their control points, the `curveToPoint:controlPoint1:controlPoint2:` method can be used to draw curves that way. All curve operations set the current point to the destination of the curve.

*Close Path Operation*
: As bezier paths actually consist of many sub-paths, one can close the current set of path operations with the `closePath` method to avoid creating a new NSBezierPath method.

A bezier path also consists of a number of *sub-paths*. After a series of move, line and curve operations, a *close path* operation is inserted to into the bezier path to indicate the end of a sub path. This concept is important with the filling commands.

### 7.5.3 Stroking, Filling and Clipping

Once you've assembled a path, you can render it in a number of ways. It can be drawn (stroked), filled, or turned into a clipping region. This is done by calling the relevant methods on the bezier path when it is ready to be used. You can stroke/fill/clip a bezier path more than once (if necessary), making the paths reusable.

A simple stroke operation is induced by calling the `stroke` operation. It causes the outline described by the path to be drawn using the current pen (which can be set on the bezier path as well).

Filling operations are induced by calling the `fill` method. It fills in, using the current background colour or pattern, the areas described by the outline of the path. Two winding rules for filling are provided: the *even-odd* and *non-zero* winding rules. These affect what areas within the path that are filled, and correspond to their PostScript definitions.

A number of convenience class methods exist for simple drawing operations, setting defaults and getting information about the current state of the drawing view, aka the *graphics state*.[2] We can call `+strokeRect:` or `+fillRect:` directly to add a new rectangle or filled rectangle to the current drawing view. The `-clipRect:` method can be used to set a smaller clipping rectangle, intersecting with the current clipping rectangle (which is set by default to be the frame of your view), just before a call to `-drawRect:` is made (see below for information about clipping paths).

---

[1] *Wikipedia* has good information on the mathematics and theory behind bezier curves/paths

[2] A graphics state is a concept inherited from GNUstep's Display Postscript heritage. In Postscript a graphics state object, or *gstate*, would contain all the information about the current colours, the current affine transform, the width to draw lines with, any fill patterns, and other such information. They could be saved onto a stack and recalled later by name. GNUstep provides a more cut down and logical implementation of similiar concepts across the NSBezierPath class and others.

### 7.5.4 Text

You can also render text within a view. For this, you use an instance of the `NSText` class, which provides advanced text rendering capabilities. It acts as a base for the text view system, which should be used where you require rich text input to your application.

### 7.5.5 Images

If you just want to display an image in your application, use the `NSImageView` class. If you want to combine it with other elements in a view (e.g. clip an image or draw on top of it), you can make use of the `NSImage` class to render an image within your view.

It is described in more detail in See [Images and Imageviews], page 47.

### 7.5.6 Affine Transformations

The skewing, rotating, translation and scaling of display objects is represented in the form of an `affine transformation`. They are encapsulated in an object of `NSAffineTransform`.

These objects store a mathematical *matrix* which describes the translation of points and objects within a coordinate system.[3] They are used internally to provide the frame and bounds transformations described earlier, and can be used in your drawing code as well. You can append transformations to the current bounds transformation, to bezier paths and even to text.

A *matrix* is a two-dimensional table of numbers. It may have any number of rows and columns, and like algebraic terms, can be multiplied and added together. We can pretend the numbers in a two by one (2x1) matrix refer to a point in the cartesian coordinate system, and manipulate them like vectors.

*Vectors* is another mathematical concept that takes numbers in pairs to describe a point in the cartesian plane. For example, the vector $(1, 1)$ can refer to the same numbered point in the cartesian coordinate system. You can also represent this point as the combination of a length (given a magnitude) and a direction (given as a rotation from the x-axis, anticlockwise). Using this, we can represent $(1, 1)$ as $(\sqrt{2}, 45 degrees)$. This representation is useful for transformation in a matrix.[4]

Matrices can be combined together to produce a new affine transform that will perform the same transformation as if all the original transformations were applied in order. You usually won't need to combine them, unless you have complicated drawing code.

## 7.6 Clipping

One concept that has been mentioned in this chapter is *clipping*. It is used extensively throughout the AppKit to control the drawing code that renders it's different graphical elements.

When drawing within a view, you often may specify points outside the visible region of your view's bounds, say to blit an image. What prevents that image from obscuring other parts of the window (and indeed the screen) is clipping. A *clip* defines what region of the screen at any one time may be drawn on.

For example, when the AppKit calls your `drawRect:` method to draw onto the screen, it first calls `lockFocus`. In this method is sets a *clipping path* defined to the frame of your view's rectangle by default, so that you do not draw outside the frame of your view by accident.

---

[3] A mathematical matrix, described here, should not be confused with the `NSMatrix` class, which is a type of control that displays cells in grid form.

[4] For those that are interested, the distance is found by applying Pythagoras' theorem to the points in the equation $x^2 + y^2 = d^2$ (where x and y are the cartesian points, and d is the distance). The angle is then found by $tany/x$, where the angle is less than 360 degrees. The transformations are the same as those used for complex numbers. It's often known as a rectangular to polar transformation, and can be performed on most good quality scientific calculators.

You can define your own clipping paths that further clip the output of your drawing code within your view. It may be a simple rectangle (as used in the case of frame clipping by the AppKit), or a complex path defined by the outline of a `bezier path`.

# 8 Event handling

The way events are handled and passed between objects in GNUstep requires special treatment. It is relatively simple, but generally not well documented as to how it works, and how it is used by default in GNUstep. Before reading this chapter, you may wish to reaquaint yourself with views (see [The view concept], page 20).

Event handling can be very complex, or very simple, depending on what your trying to handle and to what extent you're using customised components. We will try to cover some of the basic concepts you may come across in this manual, as well as give a better guide to working with NSResponder and NSEvent.

We start with the target/action paridigm (which is used to implement outlets/actions in interface files), and then explain the AppKit's underlying event handling model, which is far more powerful and of interest if you are implementing your own views. It's also relevant to understanding how events are passed around in GNUstep (and a recommended read).

## 8.1 The Responder Chain

A *responder* is an object inheriting from `NSResponder`. It defines methods that are overrided by subclasses for receiving events, from simple things such as mouse clicks and keyboard presses, to more abstract events such as text selection or text modification. `NSView` inherits from `NSResponder` (and in turn `NSControl` inherits from `NSView`) so in effect, all views and controls can respond to events.

Responders are linked together in a chain, whereby a top-level graphical element (usually a window) receives an event, and if it doesn't understand it, it passes it on to higher-level graphical elements, namely views. As views can be placed inside each other, a low-level superview may pass on higher-level events to it's more abstract children. The responder chain is the programmatic linkage between different objects. It is usually setup by GNUstep, but can be modified by the programmer.

The object at the top of the focus stack in a window is usually the *first responder*, meaning that any events will be forwarded to it first, and then along the chain if necessary. You can retrieve the first responder in a window by calling `-firstResponder` against the `NSWindow` object.[1].

More than responder chain may exist, but only one may be active at a time. It is called a chain, due to the way event messages are passed through successive calls to each consecutive object in the chain.

## 8.2 Being a responder

A responder inherits the `NSResponder` class. As `NSView` inherits from this, all high-level graphical elements, including all controls and views are considered to be "responders". This class contains a number of methods for maintaining the the responder chain and default methods for handling certain types of events, such as keyboard, mouse and "text" events (for text-processing classes such as NSText).

The first method to override is `-acceptsFirstResponder`, which returns a boolean indicating whether your class will accept first responder status. You can also override `-becomeFirstResponder` and `-resignFirstResponder` to be notified of when your class gains and loses the first responder status (respectively).

The next thing to do is override the different event messages that are predefined in NSResponder, such as `-keyDown:`, `-mouseDragged:`, `-helpRequested:`, etc. What all these have in common is that they take a single `NSEvent` object argument, which contains information about the event.

[1] NSWindow objects are responders as well

Action messages are messages that have a predefined syntax i.e. they take one object as a parameter, but the name of the method that implements them defines the message. These are passed along the responder chain until a responder implementing that action message is found. This is aided via the `-tryToPerform:with:` method, which is used by GNUstep to traverse the responder chain and find an object that can perform the *anAction* selector with *anObject* as a parameter.

Some of the common ones include:

- -keyDown:
- -keyUp:
- -mouseDown:
- -mouseUp:
- -mouseMoved:
- -mouseEntered:
- -mouseExited:
- -rightMouseDown:
- -rightMouseUp:

You can also pass your own custom selectors along responder chains, trying to find the first object that responds to a particular method name. Given an object and a selector, call `-tryToPerform:with:` on an object in the responder chain, and this method will be tried on each successive responder until one can be found that responds to the selector. If a method cannot be method, it returns `NO`.

## 8.3 Target/Action Paridgm

Controls use the target/action paridgm for simple events, which only have a sender and a target.[2] The *target* object is the object notified of an event. It is like a *sink* in OLE/COM programming and is referred to as the *receiver*. The *action* is an event being performed, and takes the form of a selector. The *sender* is the object generating the action. An action is passed along the responder chain until it is processed or until the end of the responder chain is reached, in which case the message is returned to the sender indicating it couldn't be processed. Messages that are passed as such events are known as *action messages*, and these events are known as `action events`.

Let us explain with a simple example. We create a button on a form as an `NSButton` that we want to inform our `AppController` object instance when it is clicked. The button object is the *sender* and the `AppController` object is the *target*. We tell the button object to call our target object using the selector `-browseForServer:`.[3] This selector is the *action*.

Many of these actions are predefined in the `NSResponder` class which is implemented by all views.

On the other hand, things such as menu items define a number of custom such as `-save:` or `-print:`, which many, but not all AppKit classes respond to. You can define your own actions for things such as menu buttons.

Using the above example of a target, sender (which we will call *myButton*) and action, we could manually setup a link between the objects as follows:

---

[2] A *paridgm* is a mode of thinking, often applied to programming. You may have heard of the "object-oriented programming pardigm" or the "functional programming" paridgm.

[3] The name of the selector is purely arbitrary, and can be anything you like. However, it must take one parameter, which is a reference to the sender object.

```
AppController* appCont;
NSButton* myButton;

// Initialisation of button and target objects

[myButton setAction:@selector(browserForServer:)];
[myButton setTarget:appCont];
```

In the above example, whenever *myButton* is clicked, it will call the `invoke:` method on the `MyButtonTarget` instance. What you see above is what Gorm.app does when you connect an action and a target.

This paradigm is used for simple event handling in classes that derive from NSControl. See see [Basic Controls], page 16 and see [Interface Files], page 11 for more information as to how this fits together.

# 9 Tableviews

A tableview is a grid of rows and columns for the display of data. In the AppKit, you use the `NSTableView` class to put a tableview in your application.

Gorm already has a palette set up for a tableview, that you can simply drag and drop onto your application window (refer to the sections on Gorm and the Gorm manual).

More involved, however, is getting data into your tableview and responding to certain tableview specific events. Both of these are implemented using objects that respond to certain protocols, as described below.

## 9.1 Columns

Instead of taking an array oriented model of a table, the `NSTableView` class uses a column-oriented model, where columns are used as objects of separate classes, `NSTableColumn`.

These can be instantiated with the `-initWithIdentifier:` method, where the identifier is an internal object, not displayed, used by the data source (see below) to identify what column data should be placed in. You can retrieve the table column's identifier from the `-identifier` method.

This column-oriented model for displaying data makes tableviews useful for information originating from a database, where columns are akin to fields, and each row represents a record in a table or query.

FIXME: How do you initialize a table's column's and it's identifiers when a nib is loaded. Do you use one of it's delegates methods to set all this stuff up?

## 9.2 Supplying Data

When you first create your tableview, Gorm puts some example data into it for you. However, you will want to put in your own data. GNUstep uses a *data source* model, which means that you need to supply an object that will provide data for your table view.

Firstly, you will need to connect the `dataSource` outlet of a tableview in Gorm to an instantiated object of a class. Create a class that will have it's objects provide data for your tableview(s), and instantiate it so that it appears as a *top level object* in the Objects pane.

Your object will need to implement an informal protocol to supply data to the tableview, namely the `NSTableDataSource` protocol (for more information on informal protocols, see the *GNUstep Base Programming Manual*).

Most of the methods in this protocol are optional. The compulsory methods, however, are called quite often, so you should optimise them so that they run as fast as possible.

The `-numberOfRowsInTableView:` method is called to determine how many rows are to be shown in your table view. This could change from time to time, and as the data in your table view changes, you will need to inform the `NSTableView` object that data has changed (usually via the `-update` method). When it performs it's update, it will use this method to determine how many rows to draw. This method is compulsory to implement, and your will experience errors if you don't implement it.

Another compulsory method is the `-tableView:objectValueForTableColumn:row:`. This is used to get the data that the table view will put in it's cells. It will pass you the table column that the cell appears in, the cell's row, and the associated tableview. The method returns an object to be displayed in the relevant cell; this object is usually a string.

Note that if the data to be displayed in a tableview changes, it has no way of knowing if and when it has changed. For this reason, you need to call `-update` manually on the tableview to force it to redisplay. Only then will it again invoke the methods above on your data source.

If you want your tableview to be editable, you will need to implement the `-tableView:setObjectValue: forTableColumn:row:` method. You use this to update your internal representation of what is displayed in the table view to the user.

Both these methods supply a reference to a tableview as well, so the same object (or objects of the same class) can be used as a data source for more than one tableview (e.g. across many documents, or for many tables in a database).

You can setup your tableview to accept drop operations. It must implement `-tableView:writeRows:toPasteboard:`. This method is called first, and is asking you to write the data in the specified rows to the specified pasteboard. This method should return *YES* to authorise the drop, or *NO* to reject it. More methods that can be implemented to provide information to the tableview during a drop operation include `-tableView:acceptDrop:row:dropOperation:` and `-tableView:validateDrop: proposedRow:proposedDropOperation:` methods. These are used to validate and accept drop operations, where the latter is called to validate a potential drop operation, the former just before a validated drop operation is about to take place.

If the data in your data source changes for any reason, e.g. the action of a user, you must notify the tableview. You could do this by calling `-reloadData` on the tableview object, which you can reference by an outlet on your application's controller class (or otherwise).

## 9.3 Delegates

Tableviews support a delegate for catching certain events in the lifetime of an `NSTableView` object. You will need to implement the informal `NSTableViewDelegate` protocol. This is different from the data source - you implement a delegate if you wish to catch tableview specific events created by the user.

## 9.4 Notifications

A tableview many post notifications to indicate changes to it (that otherwise are not sent to the delegate) such as user selection changes or the reordering of columns or rows.

# 10 Outline Views

An *outline view* is a specialised form of table view designed for displaying hierachical data in a tree like format. It looks alot like a Windows' TreeView Control, but operates differently, provides much more powerful functionality and it is less tedious to programme.

The node's in the outline view can be collapsed and expanded and display a list of sub-nodes. This makes the outline view hierachical.

It uses the `NSOutlineView` class, which inherits from `NSTableView`. This means that most of the behaviour that applies to tableviews also applies to outline views, such as changing columns/rows and their display, etc. It extends the tableview with a hierachial layout for data, in which nodes can be expanded and contracted.

Like the table view, the outline view control uses a data source object to get it's data, as well as a delegate to modify it's behaviour. These are objects implementing the informal protocols `NSOutlineViewDataSource` and `NSOutlineViewDelegate`. Although a delegate object is optional, outline views require a data source object.

See the *GNUstep GUI Reference* for more information about outline views (including class documentation).

## 10.1 Using a Data Source

The data source for an outline view implements the `NSOutlineViewDataSource` informal protocol. Some of it's methods are compulsory; some are not.

Note that a parameter in many of the delegate's methods is an untyped object *item*. This object is supplied by you, and the outline view passes it back to your delegate as a representation of a node or leaf row.

The outline view requires you implement the following methods:

`-(id) outlineView:(NSOutlineView*)outlineView child:(int)index ofItem:(id)item`
: Returns the item that is the child of *item* at *index*. A `nil` item means that you should return the children of the root item.

`-(BOOL) outlineView:(NSOutlineView*)outlineView isItemExpandable:(id)item`
: Returns whether *item* is expandable.

`-(int) outlineView:(NSOutlineView*)outlineView numberOfChildrenOfItem:(id)item`
: Returns the number of child items of *item*.

`-(id) outlineView:(NSOutlineView*)outlineView objectValueForTableColumn:(NSTableColumn*)tableColumn byItem:(id)item`
: Returns the data object for *item* in *tableColumn* of the table view.

Full defintions of these (and optional methods) can be found in the *GNUstep GUI Manual*.

# 11 Matrix Controls

Matrix controls are groups of cells, arranged in a table like format, with each cell indexed by row and column. It is similar to a table view, but differs in that it doesn't have a predefined cell class. Instead, it uses objects of classes derived from NSCell to be implemented. A matrix is implemented by the NSMatrix class.[1]

It takes it's implementation from NSControl, and hence what applies to controls also applies to matrixes. This matrix control is used to implement browsers as well. Note that it does not use a delegate or a data source to display or obtain it's data or size. Instead, this is done using accessor methods, making the NSMatrix class more passive in it's representation of data.

## 11.1 Creating Matrix Controls

A matrix control can be creating by new `NSMatrix` instance and then calling `-initWithFrame:mode:prototype:numberOfRows:numberOfColumns:` or `-initWithFrame:mode:cellClass:numberOfRows:numberOfColumns:`. The former method uses an instance of a cell to instantiate cells for the rows and columns, while the latter uses a cell class to create the cells.

Both these methods require a matrix mode, those of which are specified in `NSMatrixMode` and control how the matrix "tracks" the mouse:

`NSRadioModeMatrix`
: Only permits one cell in the matrix to be selected at a time.

`NSHighlightModeMatrix`
: Performs trackng (as described below) as well as highlighting the cell before tracking commences.

`NSListModeMatrix`
: Cells objects are highlighted without the opportunity to track the mouse.

`NSTrackModeMatrix`
: The cell is able to track the mouse while the cursor is within it's bounds.

For more information about cell tracking, See [Basic Controls], page 16.

## 11.2 Using Matrix controls

After having placed one on a window using Gorm, we can change what appears in the matrix by using it's methods.

The *cell class* is what class is to be used (by default) to create cells. We can use instances of many different cells classes, for example, you may choose to populate your matrix with `NSTextCell` instances as well as `NSButtonCell` instances if you were creating an interactive form. You set the default cell class by calling either `+setCellClass:` on the `NSMatrix` class to set the cell class over all new `NSMatrix` instances, or you can call `-setCellClass:` to set the class used to create new cells on a instance-by-instance basis for each of your matrix instances.

We can retrieve information about the cells in a matrix through a variety of methods. To retrieve the cell at a certain location, use the `-cellAtRow:column:` method. The size of cells is retrieved using the `-cellSize` method. To access specific cells, use `-cellAtRow:column:`, or to access all the cells, simple call `-cells` to get an array.

We can begin adding rows or columns to the end our matrix using the `-addColumn` and `-addRow` methods. To specify the specific cells, use the `-addColumnWithCells:` and `-addRowWithCells:`

[1] Matrices, as referred to here, are not to be confused with affine transforms, the latter of which is commonly referred to as a matrix, due to it's internal implementation of a mathematical matrix.

methods, passing an array of the cells for that column/row. Alternatively, rows and columns can be inserted at arbitrary locations using the `-insertRow:` and `-insertColumn:` methods, specifying a row or column number. `-insertRow:withCells:` and `-insertColumn:withCells:` lets you pass in the cells to be inserted.

Rows and columns can also be removed or replaced. You can remove a column or a row by number using the `-removeColumn:` or `-removeRow:` methods respectively. To replace a particular cell, use the `-putCell:atRow:column:` method.

The cell selection and selection behaviour can be modified. A specific cell can be selected with the `-selectCellAtRow:column:` by specifying it's location, `-selectCellWithTag:` by specifying it's tag, or `-selectCell:` with the cell object. You can also select all the cells with the `-selectAll:` method.

The selected cell is returned from `-selectedCell:`, or `-selectedCells` if more than one cell is selected. `-selectedRow` and `-selectedColumn` can be used if an entire row/column is selected.

# 12 Browsers

A *browser* is a special type of matrix control, useful for the display of hierachial or tree-like data. They use vertical lists of cells, in which some cells can be selected so that they display the "branches" of a tree in the adjacent pane. In this way, a user can easily navigate a hierachy, such as a filesystem which has many directories and sub-directories.

In fact, the textual data in a browser can be accessed using path like string components, such as '`/path/to/leaf`' or '`/usr/local/lib`'. A good example of it's use in filesystems is GWorkspace, GNUstep's file manager application.[1].

We introduce the concept of *leaves* and *branches*. A *leaf* is a simple browser cell that only display's text; it does not open a new browser pane with sub-cells when it is selected. A *branch* both display text, and when selected, it fills the pane to the right with a list of leaves and/or branches that represent a group of cells logically below this one. A branch shows an arrow to indicate that it can be selected to display sub-cells. It is useful when dealing with tree-structures such as that modelled in Computer Sciencei courses.

Each pane in the browser view is actually a one-column matrix (an `NSMatrix` object) which can be returned.

Like many other controls, browsers define their own cell class, known as `NSBrowserCell`. It provides methods that are used to implement the functionality described above. Browsers use a simple delegate to decide how to display your hierachial data, which can be passive or active (see below).

## 12.1 Browser Cells

As mentioned above, `NSBrowserCell` is used to implement the cells's placed in a browser. As a class it is quite simple, and warrants little attention.

It responds to all the methods in `NSCell`, such as `setText:` and the set value methods. Additionally, we can find out if it is a leaf using the `-isLeaf` method, and set whether or not it is a leaf using the `-setLeaf:` method.

You can set whether the browser cell is selected using the `-set` method, and reset it using the `-reset` method. A cell shows that it is selected (or "set") when it is highlighted.

## 12.2 Browser Methods

Browsers provide a number of methods used for customising their behaviour, setting their data and getting information about their state.

The path to the currently selected item (as described above) can be found using the `-path` method. You can find out the path leading upto a column with the `-pathToColumn:` method. An easy way of setting the current path is the `-setPath:` operator.

You can customize the appearance of a browser and it's columns in various ways. Use `-setSeparatesColumns:` to have each column drawn in a separate pane. `-setTakesTitleFromPreviousColumn:` has it take the title displayed in the current column from the cell selected in the previous column, while `-setTitle:ofColumn:` allows you to set a column title directly. `-setTitled:` changes whether column titles are displayed at all.

The types of operations permitted by the user can be changed as well. `-setAllowsMultipleSelection:` can be used to allow multiple selection, while `-setAllowsEmptySelection:` can be used to permit nothing to be selected. Use `-setAllowsBranchSelection:` to allow multiple branches to be selected when in multiple selection mode.

[1] Note that GWorkspace customises it's browser controls significantly

The first and last column visible in the browser is found via the `-firstVisibleColumn` and `-lastVisibleColumn` respectively.

## 12.3 Browser Delegate

The delegate for a browser is used to gather it's data. It can be optionally *passive* or *active*, the difference being that active delegates instantiate the browser cell's themselves, whilst passive delegates leave this to `NSBrowser`. As a result, you can only implement one or the other subset of methods in `NSBrowserDelegate` informal protocol..

A *passive* delegate must implement the `-browser:numberOfRowsInColumn:`, returning the number of rows to appear in the specified column number. On the other hand, *active* delegates must implement `-browser:createRowsForColumn:inMatrix:` and create the cells for that column proactively. You can only implement one of these methods; not both.

All browser delegates can implement `-browser:willDisplayCell:atRow:column:`, a method called by the browser object before a particular cell is displayed so that the delegate can set up its properties. This method is a must for passive delegates. Another method that should be implemented is `-browser:selectRow:inColumn:`, as it is the delegate's responsibility to select cells (often by calling `-set` on the corresponding `NSBrowserCell` object). This method returns whether or not the cell was selected.

You can optionally implement a number of other delegate methods if you wish. `-browser:titleOfColumn:` is called to get the title for a certain column, returned as a string, before the column is drawn.

Keeping track of when the browser scrolls can be accomplished by implementing the `-browserWillScroll:` and/or the `-browserDidScroll:` methods. You can also specify to the browser whether or not columns are "valid" by implementing the `-browser:isColumnValid:` method. This is called by the browser in response to its `-validateVisibleColumn:` method, which checks whether a column is invalid and needs redrawing.

# 13 Data Exchange

*Data Exchange* refers to the many high-level options GNUstep provides for allowing different applications to exchange common types of data. The sorts of services include "cut and paste", "drag and drop", service applications, filter services and distributed objects.

We begin our discussion with an explanation of pasteboards, which form the basis of data exchange in GNUstep. We will then go on to explain how your application can expose or consume these different sorts of data exchange services. However you receive data, it will most likely involve the use of pasteboards, hence the next section is very important.

## 13.1 Pasteboards

A *pasteboard* is the helper object used to exchange data between applications. It is an instance of the `NSPasteboard` class. Data is written to the pasteboard in different forms that it can be represented, so that the application or service receiving the data can use it.

There is a pasteboard server, a service provided with GNUstep which handles pasting between GNUstep applications. You may recognise it as the `gpbs` application.

Every pasteboard has a name that can be used to identify it. This is a string, which should be unique, but some standard pasteboard names are defined for certain uses:

`NSGeneralPboard`
: The *general pasteboard*, often used with copy and paste.

`NSFontPboard`
: Used for the exchange of font data.

`NSRulerPboard`
: Used for the exchange of ruler data.

`NSFindPboard`
: Used for "Find and Replace" editing.

`NSDragPboard`
: Used in the exchange of drag'n'drop data.

You can retreive a pasteboard by name using the `+pasteboardWithName:` method, or with a guaranteed unique name with the `+pasteboardWithUniqueName` method.

All pasteboards also have any number of *types*. A *type* is simply one form of data that the pasteboard will contain, such as HTML data or text data. The supported data types are listed below:

- NSColorPboardType
- NSDataLinkPboardType
- NSFileContentsPboardType
- NSFilenamesPboardType
- NSFontPboardType
- NSGeneralPboardType
- NSHTMLPboardType
- NSPostScriptPboardType
- NSPDFPboardType
- NSPICTPboardType
- NSRTFPboardType
- NSRTFDPboardType

- NSRulerPboardType
- NSStringPboardType
- NSTabularTextPboardType
- NSTIFFPboardType
- NSURLPboardType

See the AppKit manual for more information about storing these types of data on a pasteboard.

Finally a pasteboard may or may not have an *owner*. An *owner* is an object implmenting the `NSPasteboardOwner` informal protocol that can provide the pasteboard with data of a certain type upon request. If you don't supply an owner object, you should store the data onto the pasteboard straight away.

### 13.1.1 Constructing a pasteboard

You can get a pasteboard using the `+pasteboardWithName:` method with one of the standard names above, or the `+pasteboardWithUniqueName:` for a pasteboard with a name that is unique to the pasteboard server. You can also get a pasteboard based on the available filter services by calling `+pasteboardByFilteringFile:` for a pasteboard containing file, accessible by all the data types that it can be filtered to. If you know the source data type, you can use `+pasteboardByFilterData:ofType:` specifying a data object for a pasteboard that can convert data to different types that can be filtered from your data's type.

If you are constructing a pasteboard, you will want to call `-setData:forType:` method to put the associated data in the pasteboard for another object to read it out. Use `-declareTypes:owner:` to declare the types that this pasteboard will contain, and an owner object that will supply the data for those types that you don't explicitly write to the pasteboard.

### 13.1.2 Using an Owner

You can provide a pasteboard owner by implementing the `NSPasteboardOwner` informal protocol. This is used for the "lazy" provision of data. The pasteboard will call methods on the owner when it can't find the data being requested already stored on it.

The first method to implement is `-pasteboard:provideDataForType:`. This is called when the pasteboard doesn't have the data specified by *type*. You give it to the pasteboard by calling `-setData:forType:` on the pasteboard.

We can also implement `-pasteboardChangedOwner:`, which informs us that the owner has been changed and we no longer have to provide data to the pasteboard. GNUstep also has an extension, the `-pasteboard:provideDataForType:andVersion:` which should be implemented when data of a certain version as well is required.

## 13.2 Cut and Paste

Cut, copy and paste is the most common service you will want to provide in your application. Thankfully, all standard GNUstep objects handle copying and pasting where commonly appropriate, such as the NSText variety of objects. However, in some cases it may be useful to provide copying and pasting services, especially for your own views or on customised GNUstep views.

The first thing to to is to provide two methods on your object called `-cut:`, `-copy:`, and/or `-paste:` both taking an object (the sender) as their first parameter. This will enable Gorm's standard "Cut", "Copy" and "Paste" menu items if you place them in your interface.

You will usually use the general pasteboard for cut and paste, which can be retreived by going:

```
NSPasteboard* generalPB = [NSPasteboard pasteboardWithName:NSGeneralPboard];
```

The implementation of these methods should then follow. For cutting and copying:

1. Create a pasteboard

   Usually we use the general pasteboard, but you can create one with your own name if you like.

2. Register types of data to be provided

   The next thing to do is specify which types of data you will provide on the pasteboard. Use the `-declareTypes:owner:` method, passing an array of types, and optionally, an owner object.

3. Provide data for pasting

   You supply data to the pasteboard for pasting by using the `-setData:forType:` method. If you have used an owner, make sure that it implements the `NSPasteboardOwner` protocol and that it can return data in the form(s) specified in the previous step.

If you decide to provide data in a number of types, it is often recommended you supply the *richest* type directly to the pasteboard, and use an owner to supply more basic data types. Simply use `if/else if` statements in the `-pasteboard:provideDataForType:` method on your owner.

Pasting data is much simpler. Simply retreive the general pasteboard, and call the `-stringForType:` or `-propertyListForType:` method, passing in a type.

Make sure that you declare the types your pasteboard supports with the `-declareTypes:owner:` method. You can specify nil for the owner if you are not using lazy data provision.

## 13.3 Drag and Drop

*Drag and drop* is often more complex. Many different standard views provide their own delegate protocols for receiving drag and drop events, and you should refer to the documentation for those (especially tableviews and outline views) before following the instructions in this section. However, this is still useful in explaining some important concepts.

Such operations consist of both a *drag* and a *drop*[1]. The *drag* occurs when the user clicks their mouse button on a visible GUI element, and begins to mouse the mouse away from it. A *drop* occurs when the user moves the mouse over another GUI element and releases the mouse button. Obviously, dragging and dropping can only occur on visible elements of the screen that take up some real estate.

Below, we discuss dragging sources and dragging destinations, and what is required to make your views responsive as such.

### 13.3.1 Dragging Sources

When a drag event is initiated, the `-dragImage:at:offset: event:pasteboard: source:slideBack:` method is called on your subclass of NSView. In this method, you need to supply a dragging image, a pasteboard to hold the data, and and a *dragging source* object (specified by the `source:` parameter).

The *dragging source* object should implement the `NSDraggingSource` protocol. The main method to implement is `-draggingSourceOperationMaskForLocal:`, whereas the others are used for dragging session events (and are otherwise optional). In this method, you should return the set of binary or-ed values corresponding to the permitted drag operations on this displayed image representation, listed below:

`NSDragOperationNone`
: No drag operations are permitted with this data.

[1] I know this seems really, really obvious, but just play along; I'm not trying to be patronising.

`NSDragOperationCopy`
: This data can be copied.

`NSDragOperationLink`
: This data can be "shared". FIXME: WTF does this mean.

`NSDragOperationGeneric`
: The type of drag operation that this is can be defined by the dragging destination.

`NSDragOperationPrivate`
: This type of drag operation is defined privately by the source and destination objects, and hence negotiated between them.

`NSDragOperationMove`
: The data represented by this drag operation can be moved to the destination.

`NSDragOperationDelete`
: The destination can be responsible for deleting the data.

`NSDragOperationAll`
`NSDragOperationEvery`
: All the above drag operations are acceptable.

You can specify more than one of the above by binary or-ing them together (the single pipe operator). Note that if you permit the `NSDragOperationMove` or `NSDragOperationDelete` methods, you must implement the `-draggedImage:endedAt:operation:` method, which is called when a dragging operation is finished so that your source can cleanup any visual or internal data in the source (such as making the source image disappear).

### 13.3.2 Dragging Destinations

A view or window that is to act as a *dragging destination* should be sent the message `-registerForDraggedTypes:` with an array of the accepted dragging types. The view or window should then implement some of the methods in the `NSDraggingDestination` informal protocol.

Some of these methods are listed below:

`- draggingEntered:`
: This is method is called when the user drags something into the frame of your window or view. Use it to return what dragging types you will permit for the dragging info passed in `sender`.

`- prepareForDragOperation:`
: This is called just after the user has dropped the dragged object. Use this method to make any preparations for the drop. Return YES to cancel the drop.

`- performDragOperation:`
: This method is called so that you can perform the drop operation. This method is a must to implement.

`- concludeDragOperation:`
: This is again optional, and can be used to perform any cleanup or post-drop operations.

`- draggingUpdated:`
: This is called periodically as the drag image is moved within your frame. It can be optionally implemented to update the drag operation with different drag types (returned) as the user moves the drag images over various parts of your view. It may be useful if drag operations are context sensitive with respect to the graphical elements that your view displays.

Hence to act as a dragging destination, you need to at least implement `-draggingEntered:` and `-performDragOperation`. Make sure that when you actually perform the drag operation, that you retreive the pasteboard being used from the dragging destination (see below), as opposed to just retreiving the `NSDragPboard` named pasteboard, as you cannot be certain which pasteboard the dragging source has used for the drag operation.

### 13.3.3 Dragging Information

Both `NSDraggingSource` and `NSDraggingDestination` use objects implementing the `NSDraggingInformation` protocol to convey information about the drag'n'drop operation. You can use this to make better decisions in many of the above mentioned methods with relation to permitting/disallowing different drag types and drag operations. You never implement this protocol, however.

The pasteboard being used for the drag operation can be retreived via the `-draggingPasteboard` method. The image being used for the drag operation and its location can be retrieved via the `-draggingImage` and `-draggingImageLocation` methods respectively. If you need to snap the image during the drag operation, use the `-slideDraggedImageTo:` method, but only do this during `-prepareForDragOperation:` in the destination object.

## 13.4 Services and Filter Services

A *service* is a special type of application or tool that can be used to process data outside of the application. An application can both take advantage of services, or provide them to other applications. Like "cut and paste" and "drag'n'drop", services use pasteboard to receive data and send it back to the calling application.

A user can usually make use of a service by selecting something in your application (such as some text or an object) and selecting a service from the "Services" menu. You can also invoke services programmatically.

One way is to put a *Services* sub-menu in your interface file's menu using `Gorm` (as mentioned above. The other way is to call the `NSPerformService()` function. It takes two parameters, a service name and a pasteboard. If the service invocation is successful, the pasteboard will contain the output data from the service. The latter method is useful for filter services (described below).

A service becomes available to any NSResponder object in your application's interface. Most GNUstep classes are setup to consume services, but if you have your own `NSView` or `NSWindow` subclasses, you will need to implement extra methods so that it can make use of services. A service that isn't available to an object will not appear available in the Services menu.

Providing services is a little bit different, and requires a bit more work. You can implement a service as a normal GNUstep application, or as a special command-line type using the `service.make` template in your GNUmakefile. Either way, you need to also provide extra information in your '`Info-gnustep.plist`' file that describes what services your application provides.

## 13.5 Providing Services

There are two types of services you can provide: normal services and filter services. Normal services may either send, receive or both send and receive data. They are often useful for initiating outside processes based on simple string information, such as loading up a "New Message" window in your email client with an email address that the user has highlighted in your application. Such as service wouldn't need to return any data. These services also are registered to appear in the "Services" menu of applications.

For example, the user would highlight an email address in a text box, and then select "Send Email" from the services menu. GNUstep would then locate the associated service and put the

email address on a pasteboard. The pasteboard is sent to the service application (and loaded if necessary), which processes it accordingly.

On the other hand, a filter service is much more specific. They are designed to convert data from one type to another, and are only ever invoked programatically i.e. they don't appear in the "Services" menu.

We begin making our application or command-line service ready for acting as a service by calling `-setServicesProvider:` on the `NSApplication` object, or by calling the `NSRegisterServicesProvider()` function. Both take an object, which will provide the service, as a parameter, and the latter also takes a *port name* as a string, which will be used to contact the application. `NSApplication` uses the name of your application as the port name.

Secondly, the object that will provide the service needs to implement a method in the form of: `[methodname]:userData:error:`, where *methodname* is a custom, arbitrary name of the method. For example if you were to create a service that encrypts data and you want to call it something like `-encryptData`, the method would take the form:

```
- (void) encryptData:(NSPasteboard*)pBoard
    userData:(NSString*)userData
       error:(NSString**)error;
```

As you can see, the first part is arbitrary, but the rest must be the same for all services. It is the first part that you will use in the `NSMessage` key below.

Lastly, all services need a special addition to their Info-gnustep.plist file, which should be included as a `RESOURCE` in your '`GNUmakefile`'. See the *GNUstep Makefile Manual* for more details.

### 13.5.1 Normal Services

As mentioned before, normal services may either send data, receive data, or both. Their Info-gnustep.plist file must have a top-level key named `NSServices`, which becomes an array of dictionaries. This array has one dictionary per service that is provided.

Each service dictionary has the following keys:

*NSMessage*
: This is the first part of the method name (as described above). For example, if your services provider object implements a method called `-randomData:userData:error:`, this key should take a string value equal to `"randomData"`.

*NSSendTypes*
: This key contains an array of the types of data that your service provider can handle (the types of data that may be sent to it). These types are the string values defined earlier for pasteboard types. Simply use the same name as used in the source code, e.g. `NSStringPboardType`.

*NSReturnTypes*
: This is an array of string values that contains the types your service provider can return.

*NSPortName*
: A service must be contacted via a distributed objects port, and this string value must contain the name of the port your application will be listening on for message. Unless you are not writing an application, this is usually set to the name of the application.

*NSMenuItem*
: This is a dictionary used to set the menu item name for this service. Each key in the dictionary is the name of a language, with the value set to a string that will

be displayed as the menu item for this service in the set language. You can also provide a *default* key, which will be displayed if none of the translations you have provided match the user's language settings. You may also place one forward slash character ('/') in the menu item name, which will be used to split the item into a sub-menu of `Services` and the menu item. It is useful for grouping related services in a sub-menu.

*NSUserData*
: This key is optional, and is set to a string value which may be whatever you like. It is passed to the method implementing the service. Use this if you want the one method to handle a number of service implementations, which are selected based on this string.

*NSKeyEquivalent*
: This is an optional dictionary which contains the key equivalents to the menu items you have listed in *NSMenuItem*. Each dictionary key is the name of a language (or *default* as described above) with its value set to a single letter that corresponds to a keyboard key.

*NSTimeout*
: This key is optional, and specifies how long the system should wait for the service provider to complete providing the service. It is a number in milliseconds. By default, the system waits 30 seconds.

*NSExecutable*
: This is an optional string value that contains the path of the executable which should be launched if the service is not already running. You will not usually need this for normal applications.

## Example

We want to provide a service that turns ordinary string data into coded HTML text. Our service application is called "WebSiteEditor" and the method that provides the HTML translation is called `textToHtml:`. It accepts string data, and publishes the HTML back in string form.

Our example Info-gnustep.plist array could be:

```
{
  ..

   (application specific keys)

  ..

  NSServices = (
    {
      NSPortName = WebSiteEditor;
      NSMessage = textToHtml;
      NSSendTypes = ( NSStringPboardType );
      NSReturnTypes = ( NSStringPboardType );
      NSMenuItem = {
        default = "Convert to HTML";
      };
      NSTimeout = 25000;
      NSKeyEquivalent = {
        default = H;
      };
```

```
        NSUserData = "NoBodyTags";
      }
      .. (More service definitions)
    );
}
```

As can be seen above, *NSServices* is an array containing one dictionary, which corresponds to one service. The service appears in the menu as "Convert to HTML", which expects string data. A possible code implementation may be:

```
- (void) textToHTML:(NSPasteboard*)pboard
           userData:(NSString*)userData
              error:(NString**)error
{
  NSString* data, *convertedData;
  if ([[pboard types] containsObject:NSStringPboardType])
  {
     // Extract string data from pasteboard
     data = [pboard stringForType:NSStringPboardType];

     // Convert to HTML as a string
     //..

     // Put the result back onto the pasteboard
     [pboard declareTypes:[NSArrayWithObject:NSStringPboardType]
                    owner:nil
     [pboard setString:convertedData forType:NSStringPboardType];
  }
  else
    *error = Ïncorrect data type provided to textToHTML: service.";
}
```

### 13.5.2 Filter Services

As mentioned before, filter services are not initiated by the user, but are initiated by programme's to convert data from one type to another. They also have entries in an application's '`Info-gnustep.plist`' *NSServices* array. These entries are dictionaries as well, but they contain the following keys:

*NSFilter* This is the equivalent of the *NSMessage* key used for normal services. It is the name of the distributed objects port that the filter service will listen on for messages. It again is usually set to the name of the application, but as filter services are more likely to be standalone tools, this one can differ somewhat.

*NSInputMechanism*
This is an optional key that specifies a string value corresponding to a different input mechanism than the usual distributed objects message passing. These values may be:

*NSIdentity*
The data is placed on a pasteboard. It is not changed.

*NSMapFile*
The data is the name of a file. The contents of this file will be placed on the pasteboard instead.

*NSUnixStdio*
: The data is the name of a file. This file is passed as an argument to a command-line programme, which is executed. The stdout of the programme is placed on the pasteboard instead.

### 13.5.3 Registering Services

Before a service can be consumed by applications, it must be registered programatically and on the command line.

An application registers the object that will be providing service(s) by calling `-setServicesProvider:` on their `NSApplication` object. Tool applications must call `NSRegisterServicesProvider()`, which is a function that takes the service object and the port name (as specified by `NSMessage` or `NSFilter` in the '`Info-gnustep.plist`' file).

Once that is in code and your application has been installed, you also need to execute `make_services`, which is a script that comes with GNUstep. It locates the Info-gnustep.plist file and builds a list of services. This list of services becomes available to applications started after the script is executed.

## 13.6 Using Services

For the most part, AppKit objects are implemented to take advantage of most service types where appropriate, especially in regard to string data. However, there are situations where you will want to register for service consumption yourself, or where you want to allow your custom views to consume services.

An *NSResponder* object must first register the pasteboard types it supports. Then, when a user tries to invoke a service, GNUstep first checks the responder chain for an object that can handle the service's input type, and then it queries the object for the data to be processed by the service. If data is returned from the service, GNUstep then gives the pasteboard back to the object for it's own processing.

1. Registering for service consumption

   Your object must at some point register it's ability to consume services of certain pasteboard types. It does this by calling `-registerServicesMenuSendTypes: returnTypes:` with an array of send types (the pasteboard types the object can send to a service) and return types (the pasteboard types the object can receive from a service).

   This method is to be only called once for your subclass. It is convenient to put it in your class' `+initialize` method, which is usually called after your class is loaded into the runtime (and hence only once).

2. Sending data to a Service

   When sending data to a service, GNUstep must first check that your object can send those types of data before it requests the pasteboard from your object.

   So that GNUstep can check whether your object is able to export the pasteboard types requested by the service, you must implement the `-validRequestorForSendType:returnType:` method. It is passed a send pasteboard type and a return pasteboard type.

   Your implementation should return an object if it is capable of handling that combination of send and return type (and is ready to do so), or return nil if it can't. It usually returns `self`.

   If GNUstep gets a positive answer to this method, it will then call `-writeSelectionToPasteboard:types` on your object. You should implement this method to fill the pasteboard with data (or use lazy provision, as discussed earlier in this chapter). It should return `YES` if it suceeds, or `NO` if it fails.

3. Receiving data from a service

   If the service being invoked returns data, GNUstep will call `-readSelectionFromPasteboard:` on your object when the service returns. This method should retrieve the service data from the pasteboard and use that data to update it's object's state.

# 14 Images and Imageviews

GNUstep provides mechanisms for the retreival and display of images. A number of objects beginning with `NSImage`[1] exist, each with slightly different functions.

An *image* is represented using an instance of the `NSImage` class. You can create these using the path or URL of a file, raw image data or a pasteboard.

Images may contain zero or more image representations, or *imagereps*. For example, a photographic image may contain both black and white and colour representations, or representations at different resolutions. The purpose of this is to allow GNUstep to select the best representation for a particular device. GNUstep may select a lower-resolution representation for a screen, while selecting the highest resolution representation for printed output. If there was also a vector representation, it may choose to use it for printed output. An imagerep is represented by an instance of a `NSImageRep` subclass.

An image by itself is not enough for rendering. Images are rendered on a window using an `NSImageView` object. These let you set the alignment and scaling for displaying the image. They also let you set a graphical border using the `-setFrameStyle:` method. `NSImageView` is a control, so the normal control/cell model applies to it.

If you only need to display an image on it's own, use `NSImageView`. For more complicated image rendering, e.g. inside of custom views, use `NSImage` to draw or composite at a certain point.

## 14.1 Using NSImage

Whether using `NSImageView` or not, you will have to create an `NSImage` object. It provides a number of constructors for loading an image with a path, a URL or a data object (`NSData`). Note that for loading from a file or URL, two sets of methods are provided. These have subtly different meanings, as shown below:

`-initWithContentsOfFile:`
`-initWithContentsOfURL:`
: These methods load the image from the specified location, and create image representations for rendering later.

`-initByReferencingFile:`
`-initByReferencingURL:`
: These methods don't actually load the image straight away. Instead, when you try to composite or draw the image at a location, it loads the image from disk and generates a representation at that time.

From here, an image can be drawn within a view using any of the drawing/compositing/dissolving methods. You can also get at the imagereps using the `-representations` method (amongst others).

## 14.2 Drawing Images

`NSImage` provides a number of methods for drawing an image. It also provides quite a number means to control how an image is composited at its destination.[2] *Compositing* refers to the way the image is rendered onto the destination surface.

Simply drawing an image into your view may be achieved with the `-drawRepresentation:inRect:` method. In other cases, you may wish to draw it onto a destination surface with a compositing operation, in which

[1] Including NSImage

[2] Note that many of the mechanisms provided for compositing may not be supported in some backends i.e. the Windows backend.

case you can use the `-drawAtPoint:fromRect:operation:fraction:` or `-drawInRect:fromRect:operation:fraction:` methods.

These take a rectangle from the source image, and composite it onto a destination surface. The compositing operation specifies how the image is blended with the destination surface, and is a constant in `NSCompositingOperation`. These constants define what the destination image looks like after a composite, as a result of combining the source and destination image.[3]

`NSCompositeClear`
: The destination is left transparent.

`NSCompositeCopy`
: The source image appears at the destination.

`NSCompositeSourceOver`
: The source image appears wherever it is opaque, and the destination image elsewhere.

`NSCompositeSourceIn`
: The source image appears wherever both the source and destination are opaque, and is transparent elsewhere.

`NSCompositeSourceOut`
: The source image appears where the source image is opaque but the destination image is transparent, and is transparent elsewhere.

`NSCompositeSourceAtop`
: The source image appears wherever both images are opaque, the destination appears wherever the destination is opaque but the source image is transparent, and the result is transparent elsewhere.

`NSCompositeDestinationOver`
: The destination image appears wherever it is opaque, and the source image elsewhere.

`NSCompositeDestinationIn`
: The destination image appears where both images are opaque, and the result is transparent elsewhere.

`NSCompositeDestinationOut`
: The destination image appears wherever it is opaque but the source image is transparent, and it is transparent elsewhere.

`NSCompositeDestinationAtop`
: The destination image appears wherever both images are opaque, the source image appears wherever the source image is opaque but the destination is transparent, and the result is transparent elsewhere.

`NSCompositeXOR`
: The result of and exclusive OR operation between the bits defining the pixels in bothe images.

`NSCompositePlusDarker`
: The result of adding the values of the colour components of the pixels in both images, with the result approaching zero as a limit.

[3] These compositing operations are the same as those described in the (now well-studied) academic paper, *Compositing Digital Images* by Thomas Porter and Tom Duff in 1984.

`NSCompositePlusLighter`
: The result of adding the values of the colour components of the pixels in both images, with the result approaching one as a limit.

The *fraction* parameter corresponds to the coverage of the source alpha channel with zero making the source transparent and one making the source fully opaque.

## 14.3 Working with image representations

Quite a number of classes inherit from `NSImageRep` to provide means to load different types of image formats, such as bitmaps, TIFF images, etc:

`NSBitmapImageRep`
: For bitmap (raster) images, `NSBitmapImageRep` is used. PNG, JPG and TIFF image file formats would be represented with the class.

  You can retreive information about the image with methods such as `-bitsPerPixel` or `-isPlanar`. For image formats that can store metadata (such as resolution information or camera settings), the `-valueForProperty:` and `-setProperty:withValue:` methods can be used to manipulate it.

  If necessary, it contains initialisers for instantiating it from raw data(`-initWithData:`) and from the display (`-initWithFocusedViewRect:`.

`NSCachedImageRep`
: This image representation is cached bitmap left over from the result of executing some instructions or data. It lives inside an off-screen window.

`NSCustomImageRep`
: These are representations which can be drawn in a manner that is defined by the application. You initialize these with a selector that is executed against a delegate object when `-draw` is called on the representation.

`NSEPSImageRep`
: Unsupported.

The `NSImageRep` class itself also provides a number of methods for gaining information about what kinds of file formats GNUstep supports, and for instantiating images dynamically based on raw image data or the contents of a file or URL.

# Appendix A GNU Free Documentation License

Version 1.2, November 2002





0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship

could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute.

However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the

combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute

the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## A.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Concept Index