Conceptos avanzados en desarrollo de software libre

Jordi Campos Miralles
Ramon Navarro Bosch
Daniel Riera i Terrén (coordinador)

XP06/M2111/01807

UOC
www.uoc.edu

Daniel Riera i Terrén

Doctor ingeniero en Informática por la Universidad Autónoma de Barcelona. Profesor responsable de las asignaturas de programación de los Estudios de Informática, Multimedia y Telecomunicación de la Universitat Oberta de Catalunya.

Jordi Campos Miralles

Ingeniero en Informática por la Universidad Ramon Llull. Profesor de la Universidad de Barcelona en los estudios de Ingeniería Informática. Investigación en informática gráfica y biomedicina en la Universidad Politécnica de Catalunya.

Ramón Navarro Bosch

Ingeniero en Informática por la Universidad Politécnica de Catalunya. Investigación en informática gráfica y biomedicina en la Universidad Politécnica de Catalunya. Actualmente en estancia internacional en la Universidad de Arhus, en Dinamarca.

Primera edición: febrero 2007
Fundació per a la Universitat Oberta de Catalunya. Av. Tibidabo, 39-43, 08035 Barcelona
Material realizado por Eurecamedia, SL

Depósito legal: B-49.943-2006

Índice

Introducción

En esta asignatura se presentan tecnologías avanzadas para el desarrollo de aplicaciones mediante el uso de software libre. Estas incluyen herramientas para facilitar la programación, interfaces gráficas de usuario, así como algunos elementos gráficos existentes, entre otros. Se incluyen también algunos conceptos importantes y metodologías para ayudar en dicha tarea de programación.

En el primer capítulo se introducen algunos de los temas en los que posteriormente se irá profundizando, así como conceptos importantes para la elección de herramientas. Igualmente, se muestran los factores a tener en cuenta al diseñar aplicaciones y los beneficios que obtendremos al aplicar ciertas metodologías de trabajo.

Seguidamente, se dedica un capítulo a la presentación del software Mono, desde la instalación de su compilador, máquina virtual y librerías en diferentes distribuciones de GNU/Linux, a la sintaxis básica del lenguaje. Se explican igualmente, el uso de clases, relaciones entre clases, delegados, eventos y algunos tipos de datos complejos. El capítulo acaba mostrando dos lenguajes alternativos, Boo y Nemerle, y las diferencias básicas de éstos con C#.

El tercer capítulo presenta una alternativa libre para la implementación de aplicaciones gráficas en Mono: la biblioteca GTK. Se muestra cómo instalarla, se explica cómo crear *widgets* y qué elementos nos ofrece.

El cuarto capítulo habla de la herramienta posiblemente más extendida para utilizar GTK+ en el desarrollo de interfaces gráficas: Glade. Se muestra como ésta gestiona el código, pero requiere posteriormente, vincularlo con el resto de código de la aplicación.

Una vez somos capaces de generar interfaces gráficas, se presentan elementos gráficos (*widgets*) avanzados en el capítulo 5.

Hasta el momento, se muestra cómo programar interfaces gráficas de usuario usando Gtk y la herramienta de diseño Glade. El capítulo seis propone el aprovechamiento de los elementos básicos del proyecto Gnome para desarrollo de aplicaciones.

El séptimo capítulo propone el uso del estándar XML para la creación y transacción de documentos por la red. Mono soporta este estándar y presenta una serie de métodos para manipulación de documentos XML.

Los materiales acaban introduciendo aspectos de bibliotecas avanzadas portadas a Mono y presentando algunas de las aplicaciones con interfaz gráfico más conocidas, desarrolladas usando Mono y GTK.

Objetivos

Los materiales didácticos de esta asignatura aportan al estudiante los conocimientos necesarios para trabajar con herramientas avanzadas para el desarrollo de software. Los objetivos que hay que cumplir al finalizar la asignatura son los siguientes:

1. Conocer algunos factores que hay que tener en cuenta al diseñar aplicaciones y los beneficios que obtendremos al aplicar ciertas metodologías de trabajo.

2. Conocer diferentes tecnologías que nos permiten crear aplicaciones usando software libre. Saberlas instalar para poder utilizarlas en la creación de nuevas aplicaciones.

3. Saber programar en C#, Boo y Nemerle, apreciando las diferencias entre éstos.

4. Ser capaz de escoger entre las bibliotecas libres disponibles a la hora de programar una interfaz gráfica de usuario y sacar provecho de las ventajas que nos ofrecen.

5. Conocer algunas de las aplicaciones más destacadas hechas con estas tecnologías libres.

1. Conceptos generales

1.1. Mono

1.1.1. Introducción

El **Proyecto Mono** se creó con el objetivo de implementar en software libre las especificaciones del Common Language Infrastructure (CLI), informalmente conocido como **entorno .NET**. Estas especificaciones fueron publicadas como estándar en la Asociación europea para la estandarización de sistemas de información y comunicaciones (ECMA-International), inicialmente por Microsoft. En concreto, se trata del estándar Ecma-334* para la definición del lenguaje C# y del estándar Ecma-335** para la definición de CLI con sus correspondencias en estándares de la International Organization for Standarization (ISO).

Webs recomendadas

Proyecto Mono: <http://www.mono-project.com>

ECMA-International: <http://www.ecma-international.org>

Microsoft: <http://msdn.microsoft.com/netframework>

Organization for Standarization (ISO): <http://www.iso.org>

*** <http://www.ecma-international.org/publications/standards/Ecma-334.htm>**

**** <http://www.ecma-international.org/publications/standards/Ecma-335.htm>**

1.1.2. Arquitectura

El CLI de Microsoft fue diseñado como alternativa al entorno Java de Sun. Ambos se basan en la estrategia de usar una **máquina virtual**. Es decir, que el código fuente de los programas no se compila al código máquina de una arquitectura de computadores existente, sino que se compila al código máquina de una arquitectura ficticia, virtual. De este modo, para ejecutar una misma aplicación en diferentes arquitecturas de computador, no es necesario volver a compilar el código fuente, sino simplemente disponer de una máquina virtual que se ejecute en cada una de las arquitecturas. El programa que simula la máquina virtual es el que interpreta el código máquina virtual que ha generado el compilador, el código objeto.

Las técnicas empleadas para implementar los programas que simulan la máquina virtual son también comunes entre el CLI y el entorno Java. La más exitosa ha sido la que traduce fragmentos del código objeto al código de la arquitectura del computador que hospeda la máquina virtual, a medida que los va necesitando, el Just In Time (JIT). De hecho, el CLI fue diseñado pensando básicamente en utilizar esta técnica que ya estaba dando buenos resultados en el entorno Java.

El código objeto de la máquina virtual, que en el entorno Java se llama Bytecode, en el CLI es llamado Common Intermediate Language (CIL).

Hasta aquí las coincidencias entre los dos entornos, pero justamente en la concepción del CIL está la principal diferencia. El CLI fue diseñado con el objetivo

de facilitar la traducción de diferentes lenguajes fuente a este código intermedio. Por ese motivo, el estándar no sólo define el lenguaje de la máquina virtual sino también la base común entre todos los lenguajes que se diseñen para esta plataforma, el **Common Language Specification (CLS)**. Para permitir la interoperabilidad entre estos lenguajes fuente, también se definió el modo en el que deben estructurarse los tipos de datos, el **Common Type System (CTS)**. Todas estas definiciones, junto a la inclusión de metadatos en el código objeto, componen el estándar ECMA-335 y son las que facilitan la existencia de varios lenguajes fuente para este entorno.

Por este motivo, en la actualidad el entorno Java cuenta básicamente con un solo lenguaje fuente, el lenguaje Java, mientras que en el CLI podemos usar numerosos lenguajes fuente, tales como C++, Perl, Python, PHP, Fortran, Cobol, Boo, VisualBasic, J# e incluso el mismo Java (algunos de ellos con adaptaciones sobre su especificación original). No sólo es posible usarlos, sino combinarlos entre si. Por ejemplo, podemos declarar una clase en C++ y crear una instancia de ésta en Python.

Aparte de los lenguajes ya existentes, para los que existen compiladores a CIL, también se han creado algunos lenguajes nuevos. Entre ellos, en el momento de diseñar el CLI, también se diseñó un lenguaje para explotar al máximo todas sus posibilidades, el **lenguaje C#**. Este lenguaje fue definido originariamente por Microsoft y publicado como estándar Ecma-334. Se trata de un lenguaje de computador imperativo de la familia de los lenguajes C/C++ al igual que Java. Las semejanzas, pues, con éste último son muchas, aunque existen diferencias importantes que en su momento Microsoft propuso a Sun para ser incluidas en Java, pero que no fueron aceptadas.

1.1.3. Temas jurídicos

El Proyecto Mono básicamente pretende ofrecer el CLI y el conjunto de herramientas de bajo nivel para poder trabajar con él, todo ello basado en los estándares definidos por el Ecma-334 y el Ecma-335.

A pesar de este enfoque inicial, actualmente también incluye algunas bibliotecas de alto nivel que implementan partes de la Plataforma .NET de Microsoft que no están publicadas como estándares. Estas bibliotecas están sometidas a patentes por parte de Microsoft, lo cual podría provocar problemas jurídicos sobre el código que las utilice y la misma puesta en práctica de éstas.

La postura del Proyecto Mono ha sido la de fomentar el uso de las puesta en práctica de los estándares Ecma-334/Ecma-335 y ofrecer alternativas a las interfaces de programación (API) que se ven afectadas por las patentes de Microsoft. Se aconseja utilizar sólo los estándares y las API alternativas, y no usar las bibliotecas de alto nivel basadas en los modelos de Microsoft.

Migración al Proyecto Mono

El motivo para implementar las API de Microsoft ha sido el de ofrecer una mejor migración a programadores actuales de la Plataforma .NET al Proyecto Mono, pero se espera que a medio plazo éstos utilicen la bibliotecas alternativas libres de patentes.

1.1.4. Implementación

Actualmente existen dos iniciativas principales para implementar el CLI en software libre: el Proyecto Mono y el Portable.NET*. Este último está enmarcado dentro del Proyecto DotGNU**, que no sólo pretende implementar el CLI, sino también ofrecer toda una infraestructura completa a más alto nivel para el desarrollo de aplicaciones abiertas que trabajen en red. Esta infraestructura es una alternativa completa a las bibliotecas de alto nivel que utiliza Microsoft, ya que no son estándares y están sometidas a patentes.

* <http://www.gnu.org/projects/dotgnu/pnet.html>
** <http://www.gnu.org/projects/dotgnu>

El Proyecto Mono ofrece el CLI (estándar Ecma-335) mediante la implementación de su máquina virtual llamada **mono**. Ésta viene acompañada de un compilador para el lenguaje C# (estándar Ecma-334) llamado **Mcs**, y de un conjunto de bibliotecas a bajo nivel incluidas en los estándares.

Aparte de estas piezas básicas, también se incluye un conjunto muy amplio de bibliotecas para manejar interfaces gráficas de usuario, aceleración 3D, componentes multimedia, bases de datos, programación distribuida, etc. Todas ellas basadas en proyectos –existentes o de nueva creación– de software libre.

Por último, se incluyen bibliotecas basadas en las API de Microsoft para facilitar la migración de programadores de la Plataforma .NET, aunque se desaconseja su uso para evitar posibles problemas jurídicos.

1.2. GTK+

1.2.1. Introducción

GTK+ es un conjunto de herramientas multiplataforma para la creación de interfaces de usuario.

Como base para el funcionamiento de las distintas herramientas, hay un conjunto de bibliotecas que trabajan a bajo nivel:

- **Glib** es una biblioteca de bajo nivel básica para GTK+ y Gnome. En ella hay implementados los enlaces a nivel de C de las estructuras de datos, la implementación de los hilos, la carga dinámica de objetos y las interfaces para los bucles de ejecución.

- **Pango** es una biblioteca especializada en la visualización de las capas de texto y el renderizado de este mismo. Pone una especial atención en la internacionalización.

- **Atk** es una biblioteca para crear interfaces con características de accesibilidad. Pueden usarse herramientas como lupas de aumento, lectores de pantalla, o entradas de datos alternativas al clásico teclado o ratón de ordenador.

El conjunto de bibliotecas de GTK+ añaden sobre estas bibliotecas un conjunto de herramientas para el desarrollo de elementos visuales, de control de interacción y eventos, de dibujo de mapas de bits y de dibujo vectorial. Todas ellas se han diseñado e implementado desarrollando vínculos a una multitud de lenguajes entre los que destacamos Perl, Python, Java y C++. A partir de éstos, que son considerados básicos para el desarrollo de aplicaciones de escritorio, se han desarrollado vínculos para Ada, Haskell, PHP y C# (que nosotros usaremos). Para poder conseguir ser una plataforma de trabajo válida para cualquier aplicación se han desarrollado implementaciones sobres plataformas de la familia Microsoft y Linux. Actualmente, la implementación sobre lenguajes como .Net y Java nos permiten poder realizar aplicaciones portables a más plataformas: MacOS, Solaris, etc.

Este conjunto de bibliotecas ha crecido con el tiempo y actualmente tenemos:

- **Gdk**: biblioteca para enlazar los contenedores gráficos con las distintas bibliotecas. Gestiona los eventos, *drag and drop*, la conexión con las X, los elementos de entrada y salida hardware. Hace de puente entre la biblioteca y los distintos elementos externos que intervienen en las aplicaciones GTK.

- **GTK**: biblioteca de elementos gráficos de trabajo. En ella tenemos los distintos *widgets* que podemos usar, los estilos de visualización de los elementos, las funciones a nivel de usuario del *drag and drop*, señales, y el portapapeles entre otros. Más adelante detallaremos los elementos sobre los que podemos trabajar.

- **GObject**: biblioteca que proporciona funcionalidades sobre los objetos y una interfaz para trabajar con ellos desde C. Tiene una implementación de los tipos básicos, un sistema de señales y distintas propiedades de objetos como, por ejemplo, la herencia.

- **GdkPixbuf**: biblioteca para trabajar con imágenes de mapas de bits. Nos permite tener un buffer de trabajo y poder hacer retoques, redimensiones y trabajar a nivel de píxel.

- **Cairo**: biblioteca para trabajar con imágenes vectoriales. Esta permite utilizar la aceleración gráfica de la tarjeta para poder visualizar imágenes vectoriales y mapas de bits mapeados en estructuras.

La biblioteca Cairo

Cairo es una biblioteca externa al proyecto que ha permitido añadir gráficos y textos de gran calidad en las aplicaciones desde la versión 2.4.

Para poder desarrollar una aplicación de escritorio, seguramente sólo nos interesará usar la biblioteca GTK. En ella encontramos la mayoría de las funcio-

nes de alto nivel necesarias para programar la interacción y visualización. La programación de eventos, *drag and drop* y control de los distintos elementos visuales son fácilmente utilizables desde libGtk.

Uno de los aspectos más importantes en la biblioteca GTK es la facilidad de adaptación visual. Toda ella se ha implementado de forma modular con lo que pueden tenerse por separado el aspecto que se quieran que tengan los distintos elementos visuales. De esta forma, el usuario fácilmente puede personalizarse el color y la forma de los botones, ventanas, listas, etc.

1.2.2. Historia

Inicialmente, esta biblioteca se creó para desarrollar la aplicación GIMP con el aspecto y funcionalidad de Motif. Al formar parte del grupo de software de GNU, empezó a ser utilizada por distintas aplicaciones que necesitaban una interfaz gráfica de usuario. Pasó a ser una de las mas utilizadas cuando el entorno de escritorio Gnome apostó por ella como base de su interfaz gráfica. Añadió muchas funcionalidades y nuevos elementos visuales personalizados a las necesidades del entorno. Gracias a esta aportación, todas las aplicaciones para este escritorio usan libGtk.

1.2.3. Qt y WxWindows

Además de GTK+, en GNU/Linux hay distintas bibliotecas gráficas para el desarrollo de aplicaciones de escritorio. Actualmente hay dos más que destacan. Qt de la empresa Trolltech, sobre la que está basado el entorno de escritorio KDE. Éste tiene una licencia GPL pero con una opción a una licencia comercial pagando a la empresa. Wxwindows es una biblioteca que nació con el ánimo de ser portable de MacOS, Linux y Windows. En 1992, ante la necesidad de desarrollar una aplicación para dos plataformas distintas, y ante la falta de una biblioteca portable, crearon Wxwindows. Con los años, y después de ganar muchos usuarios, se ha portado a GTK para poder utilizar toda la potencia de este último.

1.3. Interfaces gráficas de usuario

1.3.1. ¿Qué son las interfaces gráficas de usuario?

Las interfaces gráficas de usuario son un mecanismo de conexión que facilita la interacción entre el ordenador y el usuario mediante un conjunto de imágenes y objetos gráficos además de texto.

Hay un conjunto de elementos (iconos, ventanas, casillas de texto, etc.) que nos permiten actuar de una manera cómoda con los programas para poder en-

viar y recibir información. Estas interfaces han evolucionado mucho y actualmente se dividen en los siguientes tipos:

- Orientadas a programas de escritorio. En este grupo tenemos gran cantidad de programas que se desarrollan actualmente. Ventanas, áreas de texto, botones, áreas gráficas vectoriales, áreas gráficas de mapas de bits, menús, etc. Sobre este tipo de interfaces hablaremos más adelante.

- Orientadas a programas web. En este tipo podemos incluir todas las aplicaciones web que trabajan siempre con el mismo conjunto de *widgets*: los definidos por HTML. La ventaja de ser multiplaforma y muy fácil de manejar le otorga la desventaja de tener una interacción reducida y una dificultad de programación elevada. Hoy en día se están desarrollando distintas iniciativas para mejorar la interfaz de usuario de la web (AJAX, XForms, etc.) pero sigue habiendo un problema de interacción. En una aplicación que requiere un relación cercana con el usuario de intercambio de datos puede provocar grandes perdidas de tiempo y funcionalidad.

- Orientadas a juegos. En este tipo tenemos interfaces gráficas muy artísticas, con técnicas 3D, efectos, etc. Hay una variedad enorme de entornos en esta clase, ya que normalmente se programan desde cero cada vez para adaptarse a las necesidades del juego.

Hay tipos más avanzados que dependen de sistemas de interfaz hardware más sofisticados, por ejemplo entornos 3D. Aquí hablaremos de las interfaces gráficas de usuario de escritorio.

1.3.2. ¿Cómo se estructuran?

Hay gran cantidad de interfaces de usuario de escritorio y cada una se estructura de manera específica. Por ejemplo, en una aplicación para Windows Forms podemos definir una aplicación donde podemos crear subventanas dentro de una principal de manera que nunca salgan de ésta. En una aplicación con GTK como elemento básico tenemos el *widget* GtkWindow, que es una ventana. Esta ventana se puede estructurar de dos formas:

- Estructurada: dividiendo la ventana con cajas contenedoras que nos permiten definir cada uno de los objetos visuales con una referencia relativa a toda la aplicación. Las divisiones que permite Gtk son en horizontal y vertical. De esta forma, conseguimos el efecto de que, al redimensionar la aplicación, la distribución de los objetos se redimensiona obteniendo un resultado visual mejor. El problema de este método es la dificultad de dividir toda la interfaz en divisiones horizontales y verticales.

- Fija: añadimos en una caja contenedora o ventana el *widget* GtkFixed. Con esta forma añadimos los distintos elementos visuales en el lugar donde quere-

mos que se vean. El problema está en redimensionar la aplicación: los distintos elementos se quedan en la posición original produciendo un efecto visual normalmente no deseable. Es mucho más fácil y rápido desarrollar con posiciones fijas, por lo que seria útil para prototipos y aplicaciones pequeñas.

1.4. *Widgets*

Widget es cualquier elemento gráfico que podamos usar en el diseño de una interfaz gráfica. Estos elementos definidos según cada API de programación nos permiten con su combinación generar cualquier pantalla.

En algunas API se pueden generar elementos propios como combinación de otros o con otras bibliotecas gráficas nativas como base de su visualización.

1.4.1. Contenedores

Los contenedores son un conjunto de *widgets* que nos permiten dividir y diseñar la forma de las ventanas. Con los contenedores se pueden organizar, repartir y alinear los distintos elementos gráficos que usaremos. Se estructuran en forma de árbol de manera que el movimiento y cambio de tamaño de un elemento afectará a todos sus hijos. Podemos destacar los siguientes contenedores:

- GtkAlignment - Caja para controlar la alineación y tamaño de la caja hija.
- GtkHBox - Caja contenedora horizontal.
- GtkVBox - Caja contenedora vertical.
- GtkHButtonBox - Caja contenedora para alinear los botones horizontalmente.
- GtkVButtonBox - Caja contenedora para alinear los botones verticalmente.
- GtkFixed - Contenedor para colocar los distintos *widgets* en posiciones fijas.
- GtkHPaned - Caja con dos partes horizontales adaptables por el usuario.
- GtkVPaned - Caja con dos partes verticales adaptables por el usuario.
- GtkLayout - Caja con barras deslizadoras.
- GtkNotebook - Caja con distintas pestañas.
- GtkTable - Caja contenedora dividida en una tabla.
- GtkExpander - Caja contenedora que puede esconder sus hijos.

1.4.2. Listado de *widgets*

Los *widgets* principales son los siguientes:

- GtkWindow - Objeto principal donde podemos añadir cajas contenedoras. Representa una ventana.
- GtkDialog - Es una ventana emergente.
- GtkMessageDialog - Es una ventana emergente con un mensaje.

El conjunto de objetos gráficos que podemos usar dentro de las cajas contenedoras son éstos:

- GtkAccelLabel - Etiqueta con aceleración de teclado.
- GtkImage - Una imagen.
- GtkLabel - Etiqueta con una pequeña cantidad de texto.
- GtkProgressBar - Barra de progreso.
- GtkStatusbar - Barra de estado.
- GtkStatusIcon - Muestra un icono en la barra de estado del sistema.
- GtkButton - Botón pulsador.
- GtkCheckButton - Botón para escoger una opción.
- GtkRadioButton - Botón para escoger una entre distintas opciones.
- GtkToggleButton - Botón de posición fija.
- GtkEntry - Entrada de texto en una línea.
- GtkHScale - Selector horizontal de un valor escalable.
- GtkVScale - Selector vertical de un valor escalable.
- GtkSpinButton - Obtener un valor real o numérico.
- GtkTextView - Área de texto con un "buffer" y distintas opciones de formateo.
- GtkTreeView - Árbol o lista de elementos.
- GtkComboBox - Selección de un elemento en una lista.
- GtkMenu – Menú.
- GtkMenuBar - Barra de menú.
- GtkMenuItem - Elemento de un menú.
- GtkToolbar - Barra de botones.
- GtkToolButton - Botón de la barra.
- GtkColorSelectionDialog - Ventana para seleccionar un color.
- GtkFileChooserDialog - Ventana para seleccionar un archivo.
- GtkFileChooserWidget - Elemento empotrable en una caja para seleccionar un archivo.
- GtkFontSelectionDialog - Ventana para seleccionar una fuente.
- GtkHSeparator - Línea separadora horizontal.
- GtkVSeparator - Línea separadora vertical.
- GtkCalendar - Muestra un calendario para escoger una fecha.
- GtkDrawingArea - Elemento para diseñar un *widget* para el usuario libre. Área de dibujo.
- GtkTooltips - Añade ayudas a los elementos.
- GtkAccessible - Añade accesibilidad a los elementos.

1.5. Usabilidad

1.5.1. Introducción

El término "usabilidad", adaptado del original inglés *usability*, recibe varias definiciones*, aunque todas ellas coinciden en el estudio del diseño de interfaces en las que el usuario pueda alcanzar sus objetivos de la manera más efectiva, eficiente y satisfactoria posible.

* <http://es.wikipedia.org/wiki/usabilidad>

Consideraremos que una interfaz es la conexión que interrelaciona dos sistemas heterogéneos.

En nuestro caso, las interfaces van a permitir la interacción entre una aplicación informática y un usuario humano. De esta forma, en el proceso de diseño de la interfaz de una aplicación no sólo es necesario incluir al usuario final, sino también el contexto en el que la aplicación va a ser usada.

Actualmente, la usabilidad ocupa un papel muy destacado en el desarrollo del software, al mismo nivel que parámetros como el rendimiento y la escalabilidad. Se popularizó con la aparición de las páginas web, con impulsores tan destacados como Jacob Nielsen*. Aunque su aplicación a programas de escritorio se ha contemplado especialmente desde la aparición de los entornos gráficos y el paradigma "What You See Is What You Get" (WYSIWYG). Este paradigma enfatiza la relación entre la simbología usada y el resultado obtenido, como medio para facilitar al usuario la interpretación intuitiva de las herramientas abstractas que utiliza. Uno de los trabajos más destacados vinculados al mundo del software libre es la *Guía para la interfaz*, de Gnome**.

* <http://www.useit.com>

** <http://developer.gnome.org/projects/gup/hig/2.0>

1.5.2. Factores

Los factores con los que se pretende medir de una forma objetiva el uso del software por parte de un usuario son su efectividad, eficiencia y satisfacción.

Como **efectividad** entenderemos la precisión y plenitud con la que se alcanza un objetivo. Seremos más efectivos si logramos obtener unos resultados más cercanos a los objetivos que nos habíamos planteado.

Por otra parte, diremos que la **eficiencia** es la relación de recursos empleados respecto a la efectividad lograda. De esta manera, entre dos métodos igual de efectivos para conseguir un mismo objetivo, consideraremos más eficiente aquel que requiera menos recursos para lograrlo.

Finalmente, consideraremos **satisfacción** la ausencia de incomodidad y la actitud positiva en el uso de un software. Un usuario puede mejorar su satisfacción respeto de un software si percibe que puede realizar su trabajo de modo más efectivo y eficiente, pero también si se siente cómodo porque le resulta intuitivo.

1.5.3. Beneficios

Los beneficios que se desean obtener al contemplar la usabilidad durante el desarrollo del software empiezan por la satisfacción del usuario al usar las aplicaciones, dado que aumenta su eficiencia a la hora de obtener sus objetivos.

Este aumento de la satisfacción y productividad de los usuarios repercute en un número menor de peticiones de rediseño y mantenimiento del software una vez que entra en producción. De la misma forma, los costes de asistencia se reducen y en conjunto obtenemos unos costes inferiores de explotación.

Además de reducir los costes de explotación, dado que la curva de aprendizaje de uso del software también es menor, los costes de formación y de documentación también disminuyen.

En conjunto, la dedicación a factores de usabilidad durante el desarrollo del software reduce los costes en fases posteriores y aumentan la calidad del producto final.

1.5.4. Metodología de trabajo

Las metodologías utilizadas para incluir los aspectos de usabilidad en el desarrollo del software contemplan la participación de los usuarios finales en la fase de diseño de la aplicación (*user testing*) en la simulación del contexto en el que se utilizaran los programas (escenarios).

Estos elementos nos ayudarán a tomar las decisiones sobre el diseño de la estructura subyacente de la información. Normalmente, se vincula la usabilidad al componente estético de la aplicación, pero éste está íntimamente unido a la concepción de la información que manipula el software. Por lo tanto, un cambio en la forma en la que se requiere presentar la información al usuario puede tener repercusiones en las fases iniciales de análisis y diseño de una aplicación.

El proceso que debe seguirse consistirá básicamente en:

1) definir las necesidades de la información,
2) definir la estructura de la información,
3) definir la presentación de la información,
4) contrastar y revisar los puntos anteriores con las pruebas de usuario.

Será necesario realizar pruebas con usuarios finales (*user testing*) mediante lo que se conoce como "prototipos de la aplicación". Es decir, el estudio de la interacción de los usuarios con simulaciones del comportamiento final del software. El diseño inicial se realiza normalmente mediante métodos heurísticos, que incluyen el análisis de expertos en un determinado tipo de aplicación.

Para realizar las pruebas con usuarios finales es muy importante determinar en qué contexto deben desarrollarse (escenario), dado que éste es un factor que influye notablemente en la percepción por parte del usuario.

La metodología de trabajo consiste en acotar las pruebas a realizar, detallando los datos que se desea recoger y su realización. El modelo usado en Gnome* puede considerarse válido para la mayoría de las aplicaciones.

* <http://developer.gnome.org/projects/gup/templates/fui_template.html>

1.5.5. Guías básicas

Como se ha visto hasta el momento, la usabilidad contempla un abanico de aspectos fuertemente vinculados al usuario y al contexto de uso, de tal forma que no es posible dar consejos genéricos, sino sólo indicaciones muy específicas para binomios usuario-contexto determinados.

Aun así, es posible establecer una guía básica presente en casi todos los trabajos de usabilidad:

- **El objetivo siempre es el usuario**. Por mucha lógica que le encontremos como desarrolladores, si el usuario no se siente efectivo, eficiente y satisfecho con la aplicación desarrollada será necesario replantear nuestra "lógica".

- **La aplicación debe vincularse con su contexto.** Es necesario usar el vocabulario y las metáforas propias del contexto (y no las relacionadas con la arquitectura informática de la aplicación).

- **La aplicación debe ser consistente.** Las operaciones y la manera de expresarlas deben ser consistentes internamente y con el software ya existente. De este modo el usuario puede reutilizar los conceptos y la lógica ya adquirida.

- **El usuario debe estar informado**. En todo momento es conveniente ofrecer al usuario información sobre lo que está aconteciendo, usando un lenguaje simple pero sin obviar los detalles.

- **La simplicidad debe predominar**. Siempre que sea posible, debe buscarse el modo más simple de expresar la información. Los parámetros de estética pueden mejorar notablemente el factor satisfacción del usuario.

- **El usuario debe conservar el control**. Por encima de las operaciones automatizadas, el usuario debe conservar el control para poder especificar qué debe efectuarse exactamente en caso de que lo crea conveniente.

- **La aplicación debe soportar los fallos del usuario**. Al trabajar con usuarios humanos, el error no tiene que ser una excepción, sino un componente más del proceso de interacción.

- **La interacción directa debe estar presente**. Cuando sea posible expresar una acción mediante la interacción directa con un elemento de la estructura de información, es conveniente ofrecer dicha opción.

- **Hay que aspirar siempre al mayor número de usuarios.** Los aspectos de internacionalización (traducción) y localización (adaptación de formatos) del software, junto la accesibilidad (acceso de personas con discapacidades), deben ser tenidos en cuenta desde las primeras fases de diseño.

2. Mono

2.1. Instalación

El paquete de software de Mono tiene muchas dependencias y subpaquetes posibles. Aquí tenemos los principales:

- Mono: compilador, máquina virtual y conjunto de clases básicas que nos permiten utilizar las funciones de la especificacin ECMA. En este conjunto de bibliotecas hay algunas añadidas por Mono y Novell, como por ejemplo el acceso a LDAP y una implementación de XML propia.
- Gtk-sharp: conjunto de bibliotecas que se usan en la programación de aplicaciones de escritorio utilizando el entorno GTK. En este conjunto está todo el código de los enlaces a las bibliotecas del paquete gtk, gnome, gnomevfs, vte, rsvg, glade y gconf.
- XSP: servidor de páginas web que permite tener conexiones https y http. Este programa sirve para poder ejecutar código aspx, ashx y asmx sin tener que instalar un servidor de páginas web del estilo de Apache. Robusto, muy útil para depurar y ligero de carga.
- Mod_mono: enlace desde el servidor de páginas web Apache a las bibliotecas de XSP para poder servir páginas web en ASP.NET.
- Monodoc: programa con la documentación de las clases de Mono. Están las realizadas por el proyecto Mono más un conjunto de bibliotecas extras de gente externa. Cuando se hace una biblioteca se puede documentar de manera que aparezca en el Monodoc automáticamente y editar desde el mismo programa esta documentación.
- Mono-tools: herramientas usadas para el desarrollo, como por ejemplo gunit.

Gunit

Gunit es el programa para la ayuda en la programación de unidades.

- Gecko-sharp: biblioteca para la creación de *widgets* de navegación web usando el motor gecko, el mismo usado en los navegadores de Mozilla.
- Gtksourceview-sharp: biblioteca para la creación de *widgets* de edición de código fuente. Con utilidades de reconocimiento de lenguajes, colores y números de línea.
- Boo: lenguaje parecido en sintaxis a Python pero con una conexión directa a las bibliotecas de Mono y a todos los lenguajes soportados para esta plataforma.
- IKVM: máquina virtual para poder ejecutar código y bibliotecas hechas en Java dentro de la máquina virtual de Mono. Este proyecto permite llamar desde Java a las bibliotecas de la plataforma Mono.

- Mono-debugger: versión de la máquina virtual para depurar el código. En esta versión, puede verse el estado de la memoria y variables en momento de ejecución. También con la ayuda de distintos IDE pueden usarse los puntos de anclaje en el código.

2.1.1. Nativo

Sin tener que compilar todas las bibliotecas y programas, la gente de Mono ha desarrollado un sistema para poder instalar Mono con un instalador en plataformas Linux. Una vez nos hemos bajado este archivo `.bin` podemos ejecutarlo directamente y nos descomprimirá e instalará todo lo necesario para ejecutar Mono, monodevelop y las bibliotecas GTK.

Instalador

Podéis bajar Mono de la página <http://www.mono-project.com/Downloads>.

2.1.2. Debian

En la distribución Debian hay paquetes para todas las opciones de la plataforma Mono. En el momento de escribir este subapartado, están en la distribución Etch y se pueden instalar usando la aplicación Synaptic o desde consola con Aptitude.

Los paquetes que se requieren son los siguientes: mono, mono-assemblies-base, mono-classlib-1.0, mono-classlib-2.0, mono-common, mono-devel, mono-gac, mono-gmcs, mono-jay, mono-jit, mono-mcs, mono-utils, mono-xsp, monodevelop, monodevelop-boo, monodevelop-java, monodevelop-nunit, monodevelop-versioncontrol, monodoc, monodoc-base, monodoc-browser, monodoc-dbus-1-manual, monodoc-gecko2.0-manual, monodoc-gtk2.0-manual, monodoc-gtksourceview2.0-manual, monodoc-manual, monodoc-nunit-manual, libgecko2.0-cil, libglib2.0-cil, libglade2.0-cil, libgnome2.0-cil, libgtk2.0-cil, libgtksourceview2.0-cil, libdbus-cil, libgmime2.1-cil, libevolution-cil, libvte2.0-cil, glade-gnome-2.

2.1.3. Fedora

En el momento de escribir este subapartado, está disponible Fedora Core 4, en la que no está incluido Mono por un problema de licencias.

En Fedora Core 5 ya está instalado Mono por defecto.

Para poder ejecutar Mono, hay distintas formas: una con Novell's Red Carpet. Una vez que está instalado el rpm de la aplicación, puedes descargarte Mono usando:

```
rug sa http://go-mono.com/download
rug sub mono-1.1-official
rug sub gtk-sharp-official
rug in mono-complete gtk-sharp
```

Web recomendada

Podéis descargar Novell's Red Carpet desde <http://www.snorp.net/files/packages/rcd-fc3>.

Si queréis instalar más programas y bibliotecas, hay una lista de los rpm disponibles en <http://www.go-mono.com/download/fedora-4-i386>

2.2. Herramientas básicas

El entorno de desarrollo de Mono incluye un gran número de herramientas básicas para el desarrollo tanto desde consola de texto como mediante interfaz gráfica. En general, la calidad de estas herramientas es buena, aunque el apartado de la documentación es el que está más retrasado.

Las herramientas básicas que se distribuyen con la plataforma Mono son las siguientes:

- mono: maquina virtual que interpreta el lenguage CIL (incluye profiler).
- mcs: compilador de C#.
- monop: utilidad para consultar la documentación abreviada mediante la línea de comandos.
- monodevelop: entorno integrado de desarrollo.
- monodoc: aplicación gráfica para visualización y edición de la documentación.
- xsp: servidor mínimo para generar páginas web desde CLI.
- mod_mono: módulo para integrarse con el servidor Apache.
- mdb: depurador de código mediante la línea de comandos.
- monodis: desensamblador de CIL.

Ejemplo de desarrollo en consola de texto

Compilar: `mcs Hola.cs`
Ejecutar: `mono Hola.exe`
Depurar: `mcs -debug Hola.cs ; mdb Hola.exe`
Profiler: `mono --profile Hola.exe`
Desensamblar: `monodis Hola.exe`
Documentación: `monop System.Console`

Ejemplo de desarrollo en entorno gráfico

- Monodevelop: herramienta para el desarrollo para poder diseñar interfícies gráficas y para poder implementar soluciones basadas en C#, boo, Java y Python.
- Gnunits: sistema para monitorizar la ejecución de tests.
- Monodoc: sistema de documentación. Incluye toda la API de Mono y puedes añadir la documentación de tus propias clases.

2.3. C# básico

El lenguaje C# está definido en el estándar Ecma-334*. Existen varias ediciones a partir de la original publicada en el 2000, basada en la propuesta de Microsoft, Hewlett-Packard e Intel. Aunque la definición hace referencia al entorno Common Language Infrastructure (CLI), C# no está vinculado a éste, y por lo tanto pueden existir implementaciones que no lo requieran.

*** <http://www.ecma-international.org/publications/standards/Ecma-334.htm>**

Este lenguaje de programación deriva de la familia de lenguajes imperativos que incluye C, C++ y Java. Es orientado a objetos, con soporte para comprobación de tipos (strong type checking), límites de los vectores (array bounds checking), uso

de variables no inicializadas y gestión de la memoria no usada (garbage collection). El espíritu es proporcionar un lenguaje de propósito general, para desarrollar aplicaciones distribuidas altamente portables entre diferentes plataformas. El mismo debe armonizar tanto con arquitecturas y sistemas operativos complejos y de gran escala como con sistemas mínimos empotrados.

En este punto, revisaremos las características básicas del lenguaje de manera breve y partiendo de la base de que disponéis de conocimientos previos en lenguajes orientados a objetos como C++ o Java.

2.3.1. Sentencias

La sintaxis de las sentencias básicas es la misma que en C++, aunque incluye alguna sentencia nueva como el `foreach`, que normalmente estaba presente en lenguajes considerados de desarrollo rápido:

- Comentarios: `//` (para una sola línea), `/*` (para múltiples líneas) `*/`
- Asignaciones: `i = a * 3;` (no permite confundirlas con la comparación `==`, ya que la comprobación de tipos indica que el resultado no es booleano)
- Alternativas: `if-else`, `switch` (permite usar `switch` con tipos no básicos, p. ej. `strings`)
- Iteracciones: `while`, `do-while`, `for`, `foreach` (aplicable a todos los objetos que implementen la interfaz `IEnumerable`)
- Saltos: `break`, `continue`, `goto`, `return`, `throw` (mecanismo de excepciones)

Ejemplo `foreach`

```
int[] a = new int[]{1,2,3};
foreach (int b in a)
  System.Console.WriteLine(b);
```

2.3.2. Tipos elementales

Al igual que otros lenguajes, C# dispone de tipos simples con los que se puede trabajar directamente con el valor y tipos compuestos a los que normalmente se accede mediante su referencia. En Java, todos los parámetros se pasan por valor, teniendo en cuenta que cualquier variable de tipo compuesto en realidad es una referencia a los datos en memoria. En cambio, en C# es posible pasar los datos tanto por valor como por referencia sean del tipo que sean.

- Tipos simples: contienen literalmente el valor y generalmente se guardan en la pila.

- `sbyte, short, int, long, byte, ushort, uint, ulong, float, double, decimal, char, bool, structs`

- Tipos compuestos: contienen la referencia a un objeto y generalmente se guardan en el *heap*.

Tabla tipos simples:

Tipo	Nombre Mono/.NET	Signo	Bytes	Rango
bool	System.Boolean	No	1	true / false
byte	System.Byte	No	1	0 .. 255
sbyte	System.SByt	S	1	-128 .. 127
short	System.Int16	S	2	-32.768 .. 32.767
ushort	System.Uint16	No	2	0 .. 65535
int	System.Int32	S	4	-2.147.483.648 .. 2.147.483.647
uint	System.Uint32	No	4	0 .. 4.394.967.395
long	System.Int64	S	8	-9E18..9E18
ulong	System.Uint64	No	8	0 .. 18446744073709551615
float	System.Single	S	4	+/-1.5E-45..+/-3.4E38 (7 sig)
double	System.Double	S	8	324 .. +/-1.7E308 (7 sig)
decimal	System.Decimal	S	12	+/-1E-28..+/-7.9E28 (28/29 sig)
char	System.Char		2	cualquier Unicode (16b)

Del mismo modo que Java, la biblioteca de C# proporciona unos tipos compuestos (clases de objetos) para encapsular los tipos simples existentes en el lenguaje. Por ejemplo, existe la clase `Integer`, capaz de contener los mismos datos que el tipo simple `int`. Estas clases se utilizan para albergar las funciones relacionadas con los tipos simples, como por ejemplo interpretar el contenido de un `string` como un valor entero.

En C#, a diferencia de Java, las conversiones entre los tipos simples y sus equivalentes compuestos son implícitas, y se conocen con el nombre de *boxing*. En ese caso, se copia el valor simple de la pila en el compuesto que se guarda en el *heap*.

En la concepción de C# se recuperaron algunas características de C++ que se habían desechado en Java. Por ejemplo, existen los tipos enumerados, los punteros y también las estructuras. El uso de punteros debe hacerse con precaución, ya que el `garbage collector` puede cambiar dinámicamente el emplazamiento de los objetos. Existen los modificadores `unsafe` y `fixed` para indicar que determinados fragmentos de datos los gestiona el programador y no la máquina virtual.

2.3.3. Entrada/salida

La manipulación de los flujos de datos de entrada y salida se realiza mediante la abstracción de las clases `stream`, al igual que en Java o C++. Aunque en C# se ha conservado la simplicidad de este último y por lo tanto la lectura y escritura básica resulta mucho más sencilla. Uno de los factores que más la simplifica es el hecho de que en C# no es obligatorio recoger las excepciones,

de forma que el código más simple puede ignorarlas suponiendo que no habrá errores.

- Trabajo con la consola: uso de la clase `Console` con los métodos `Read`, `ReadLine`, `Write`, `WriteLine`, `SetIn`, `SetOut` y `SetError`.
- Trabajo con archivos: uso de la clase `File` con los métodos `Exists`, `Delete`, `Copy`, `Move` y `GetAttributes`.
- Archivos de texto: métodos `OpenText`, `CreateText` y `AppendText`.
- Archivos en binario: uso de las clases `FileStream`, `StreamReader` y `StreamWriter`.
- Trabajo con el sistema de archivos: uso de la clase `Directory` y `DirectoryInfo` con métodos como `GetCurrentDirectory` o `GetFiles`.

Ejemplo manejo de archivos de texto

```
using System;
using System.IO;

public class ArchivosTexto {
  public static void Main(string []args){
    Console.Write("Nombre del archivo: ");
    string nombre = Console.ReadLine();
    Console.WriteLine("\tContenido del archivo {0}:", nombre) ;

    StreamReader leer= File.OpenText(nombre);
    string lin = leer.ReadLine() ;
    while (lin != null ) {
      Console.WriteLine(lin) ;
      lin = leer.ReadLine() ;
    }
    leer.Close() ;

    StreamWriter escribir= File.AppendText(nombre);
    escribir.WriteLine("Añadiendo contenido");
    escribir.Write("al archivo {0}\n", nombre);
    escribir.Close();
  }
}
```

Ejemplo manejo de archivos binarios

```
using System;
using System.IO;

public class FitxersStream {
  public static void Main(string []args){
    Console.Write("Nombre del archivo: ");
    string nombre = Console.ReadLine();

    FileStream leer = new FileStream(nombre,
                            FileMode.OpenOrCreate,
                            FileAccess.Read,
                            FileShare.Read );
    while (leer.Position < leer.Length){
      Console.Write(( char )leer.ReadByte()) ;
    }
    leer.Close();

    FileStream fit = new FileStream(nom,
```

```
                FileMode.OpenOrCreate,
                                FileAccess.Write,
                                FileShare.None );
    StreamWriter escribir = new StreamWriter(fit);
      escribir.BaseStream.Seek(0, SeekOrigin.End);

      escribir.WriteLine("Añadiendo contenido");

      escribir.Close() ;
    }
  }
```

2.3.4. Cadenas

El tratamiento de las cadenas de caracteres en C# sigue la tendencia a la simplificación que se inició con `string` en C++. Aunque incorpora más flexibilidad sintáctica normalmente presente en lenguajes de programación rápida. Podemos observar este factor en el ejemplo de código C# para la inicialización de cadenas, en el cual, entre otros detalles, vemos la posibilidad de usar literales de cadenas de caracteres tipo `verbatim`. Es decir, que se respeta el formato de su contenido incluso para caracteres especiales y saltos de línea.

Ejemplo de inicialización de cadenas

```
using System;

class CadenesDeclaracio {
  public static void Main() {
    String may = "llamada automtica al constructor";
    string min = String.Format("{{ Num={0,2:E} }}", 3);

    Console.Write("Sinnimos String({0}) = string({1})\n",
                  may.GetType(), min.GetType());
    Console.WriteLine(min);

    string s1 = "Hola", s2 = "Hola";
    Console.Write("Contienen lo mismo:{0}\n", s1.Equals(s2));
    Console.WriteLine("La misma referencia inicial: {0}",
                  Object.ReferenceEquals(s1,s2));
    s2 = "Adis";
    Console.WriteLine("Han canviado las refs: {0}",
                  Object.ReferenceEquals(s1,s2));
    Console.Write("Operador ==: {0}\n", s1==s2);

    string verba = @"con ""verbatim"" nos ahorramos
escapar determinados cdigos (\n, \t, \0...)
y podemos incluir ms de una lnea";

    Console.WriteLine(verba);
  }
}
```

Dado que en C# existen las propiedades de objeto y su acceso indexado, la sintaxis para acceder a uno de sus caracteres es tan sencilla como si se tratara de un vector de caracteres: por ejemplo `s[3]`. Del mismo modo, con-

Las propiedades de objeto en C# se tratarán en el subapartado 2.4.

sultar sus atributos deja de expresarse como una llamada a un método de acceso y se expresa como una `property`: por exemplo `s.Length`.

El resto de las operaciones sobre cadenas se realiza mediante métodos como, por ejemplo:

- Concatenación:
  - `String.Concat("Hola", " mundo")`
  - `s1+s2 // en C# es posible sobrecargar operadores como en C++`
  - `String.Join(" ", {"Hola","mundo"})`
- Subcadenas:
  - `"Burbuja".EndsWith("uja")`
  - `"Burbuja".StartsWith("Bur")`
  - `"Burbuja".IndexOf("rb")`
  - `"Burbuja".IndexOfAny({'a','o'})`
  - `"Burbuja".Substring(2,4)`
- Creación de nuevas cadenas:
  - `"Burbuja".Insert(1,"al")`
  - `"Burbuja".Replace('u','o')`
  - `"Burbuja".ToLower()`
  - `"Burbuja".Trim({'a','o'})`
- Expresiones regulares: mediante los métodos de la clase `System.Text.RegularExpressions`

De la misma forma que Java, los *strings* de C# son inmutables, es decir, que una vez que han sido inicializados ya no es posible cambiarles el contenido (las operaciones `Insert`, `Replace`, `ToLower`, etc. crean nuevos *strings*). Por este motivo, si se necesita trabajar con cadenas que se puedan alterar, se dispone de la clase `System.Text.StringBuilder`.

2.3.5. Vectores y tablas

El lenguaje C# conserva la distinción entre vectores unidimensionales y tablas multidimensionales. Al igual que Java, en C# se pueden crear matrices a base de insertar recursivamente vectores unidimensionales dentro de otros vectores. Esto permite crear matrices regulares pero también matrices irregulares tanto en tamaños como en tipos de datos contenidos. Pero si simplemente deseamos una matriz regular con datos homogéneos, podemos crear una tabla multidimensional, que es más eficiente en memoria y tiempo.

La sintaxis de los vectores unidimensionales es básicamente la misma que en Java:

- Definición: mediante tipos anónimos
  - `int [] a;`
- Inicialización:
  - `int [] a = new int[10];`

- `int [] a = {1,2,3}; // 3 elementos, índices del 0 al 2`

• Manipulación:

- longitud: `a.Length`
- copia: `Array.Copy, a.CopyTo`
- búsqueda: `Array.BinarySearch, a.IndexOf`

• Manipulación: `a.Reverse`, `Array.Sort`

• Recorrido: usando `foreach`

Si creamos matrices a base de vectores unidimensionales, la sintaxis también es la misma que en Java:

```
• int [][] matriz;
• matriz = new int [2][3]; // inicializamos a 2x3 regular
• matriz = new int [3][]; // o inicializamos solo las "primeras" dimensiones
• niu[1] = new int [7]; // y posteriormente las "siguientes"
• niu[1] = {3,6,4,1,8}; // también podemos hacerlo con literales
```

Por último, si creamos las matrices con las tablas multidimensionales la sintaxis se simplifica:

```
• int [,] matriz = new int[2,3]; // declaración de 2 dimensiones
• int [,] matriz = {{1, 2, 3}, {4, 5, 6}}; // literales
• matriz[1,1] = 7; // acceso a los elementos
```

A parte de los vectores y tablas, la biblioteca de C# también proporciona un conjunto de clases para trabajar con colecciones de elementos al estilo de la STL de C++ y las Collections de Java.

2.4. C# medio

En este segundo subapartado sobre C# se exponen sus posibilidades como lenguaje de programación orientado a objetos. En general, su sintaxis simplifica la disponible para lenguajes como C++ o Java, pero sin perder control, incluso añadiendo algunos modificadores nuevos. En muchos casos, en la definición del lenguaje se optó por definir modificadores por defecto de forma que una persona que se inicie en la programación, que no conoce todos los conceptos asociados con los atributos, no está forzada a hacer uso de ellos como sucede en otros lenguajes.

2.4.1. Definición de clases

Al definir las clases en C# especificaremos en primer lugar los atributos y en segundo lugar los modificadores de éstas.

Los atributos no se refieren a los miembros internos de la clase, sino que se trata de una forma flexible de poder extender los modificadores de ésta. Por ejemplo, los modificadores que tenemos a disposición son:

- `public`: accesible desde todo el resto del código.
- `private`: accesible sólo desde el mismo `namespace`.
- `internal`: sólo accesible desde el mismo `assembly`.
- `new`: para poder usar el mismo nombre que una clase heredada en clases declaradas dentro de otras.
- `protected`: acceso desde la clase contenedora y sus derivadas.
- `abstract`: para clases que no se pueden instanciar, deben heredarse desde una nueva clase.
- `sealed`: para indicar que no se puede derivar.

Y, con los atributos, podemos usar los disponibles en las bibliotecas estándar, por ejemplo:

- `Obsolete`: para que el compilador indique que dicha clase no estará disponible en las bibliotecas a medio plazo. De hecho, incluso podemos pedir al compilador que se niegue a compilar si se utiliza alguna clase obsoleta.

- `Serializable`, `NonSerialized`: para especificar si la clase puede serializarse a una cadena de caracteres para ser almacenada o transmitida.

O bien, definir nuevos atributos según nuestras necesidades de gestión del código derivando directamente de la clase: `System.Attribute`.

Las clases pueden declararse dentro de otras clases de modo que las internas pueden acceder a los atributos de sus contenedoras. En lenguaje C# es posible instanciar una clase interna sin necesidad de que exista una instancia de la externa.

2.4.2. Definición de atributos internos de clase

En el diseño de C# se tuvo en cuenta la carga que le supone al programador normalmente la definición de los métodos de acceso a los datos de ésta. Con esta idea, se distinguen los siguientes atributos internos de la clase:

- `Fields`: se corresponden a los habituales campos internos de la clase pensados para que sean accesibles sólo por el código de ésta. Pueden llevar los siguientes modificadores:

– `static`: para indicar que son datos asociados a la clase, no al objeto, comunes entre todas sus instancias y disponibles aunque no exista ninguna de ellas.

– `readonly`: sólo se le puede dar valor una vez; si se intenta de nuevo, se generará un error.

– `const`: el valor se asigna en tiempo de compilación, por lo tanto solo se pueden usar con tipos valor no referencia.

• `Properties`: se corresponden a la sintaxis pensada para la visión de los campos desde el exterior del objeto. Se permite declarar sus métodos de acceso vacíos en caso de que no haya que efectuar ninguna acción en especial.

– Declaración mínima: `public int Property{ get{} set{} }}`

– Acceso desde el exterior de la clase: `obj.Property=5; a=obj.Property`

Por lo tanto, en C# los métodos de acceso no se utilizarán como el resto de los métodos, sino que su sintaxis de uso se simplifica como si fueran campos internos públicos. Esta simplificación es sólo sintáctica, ya que la premisa de aislar los campos internos de su visión desde fuera del objeto continúa cumpliéndose.

Otro detalle introducido en C# es el acceso a los datos de una clase con una sintaxis equivalente al uso de tablas:

• `Indexers`: métodos para proporcionar un acceso a los datos con una sintaxis de tabla.
– Declaración: `public int this[int i]{ get{...} set{...} }`
– Acceso desde el exterior de la clase: `obj[3]=5; a=obj[5]`

```
using System;

public class ClasesDeCampos {
  int field   = 0;
  int []tabla = {1, 2, 3, 4};

  public int Property {
      get { return this.field; }
      set {
        if (value > 0)
          this.field = value;
      }
  }
  public int this[int i] {
      get {
        if (i>=0 && i< tabla.Length)
          return this.tabla[i];
        else
          return 0;
      }
      set {
        if (i>=0 && i< tabla.Length)
          this.tabla[i] = value;
      }
  }
  static void Main() {
```

```
        ClasesDeCampos c = new ClassesCamps ();

        Console.WriteLine("Field: " + c.field);

        Console.WriteLine("Property: " + c.Property);
        c.Property = 5;
        Console.WriteLine("Property(5): " + c.Property);
        c.Property = -5;
        Console.WriteLine("Property(-5): " + c.Property);

        Console.WriteLine("Indexer[2]: " + c[2]);
        Console.WriteLine("Indexer[9]: " + c[9]);
        c[2] = 5;
        Console.WriteLine("Indexer[2]: " + c[2]);
        c[9] = 5;
        Console.WriteLine("Indexer[9]: " + c[9]);
      }
    }
```

2.4.3. Métodos

La declaración de métodos, al igual que las clases, puede llevar asociada varios atributos y modificadores:

- Atributos: `Conditional`, `Obsolete`, etc.

- Modificadores:

`virtual`, `override` y `new` se explican en el subapartado 2.4.4, dedicado al polimorfismo.

– `abstract`: método vacío que debe implementarse en las clases derivadas.
– `static`: asociado a la clase, no se necesita ninguna instancia para usarlo, y por lo tanto no puede usar los datos internos de un objeto.
– `extern`: definido externamente (generalmente en otro lenguaje).

- `[DllImport(''util.dll'')] extern int met(int s);`

– `virtual`, `override`, `new`.

```
#define DEBUG
using System;
using System.Diagnostics;

public class EjemploAtributos {

  [Obsolete("Usad el método Nuevo", false)]
  static void Antiguo( ) { Console.WriteLine("Antiguo"); }

  static void Nuevo( )   { Console.WriteLine("Nuevo"); }

  [Conditional("DEBUG")]
  static void Depura() { Console.WriteLine("Depura"); }

  public static void Main( ) {
    Antiguo( );
    Depura();
  }
}
```

Además, la declaración de los parámetros conlleva algunas novedades respeto a sus modificadores:

- `ref`: indica que el parámetro se recibe por referencia, de modo que si cambiamos su contenido cambiará el de la variable de entrada. Dentro del cuerpo del método, el parámetro se usará sin ningún modificador especial, como en C++.

- `out`: igual que `ref` pero el compilador se asegurará de que modificamos su contenido antes de finalizar el método.

- `params`: indica que el resto de la lista de parámetros es un número indeterminado de argumentos del tipo indicado.

Como en el resto de los lenguajes orientados a objetos de la misma familia, podemos sobrecargar los métodos usando el mismo identificador con diferente lista de parámetros.

A diferencia de Java, en C# se permite sobrecargar los operadores tal y como se podía hacer en C++.

```
using System;

public class EjemploMetodos {
  static void Entrada   (    int e) { e = 1; }
  static void Salida    (out int s) { s = 2; }
  static void Referencia(ref int r) {
    Console.WriteLine("Refer. r="+r); r = 3; }

  static void Sobrecarga(out int  x){ x=4; }
  static void Sobrecarga(out char y){y='B';}

  private int f=5;
  static int operator+(EjemploMetodos a,
                       EjemploMetodos b) {
    return a.f+b.f;
  }

  static void Main() {
    int e=10, s=20, r=30, n=40;
    char c='A';

    Console.WriteLine("e({0}) s({1}) r({2})",
                      e, s, r);
    Entrada(e); Salida(out s); Referencia(ref r);
    Console.WriteLine("e({0}) s({1}) r({2})",
                      e, s, r);

    Console.WriteLine("n({0}) c({1})", n, c);
    Sobrecarga(out n); Sobrecarga(out c);
    Console.WriteLine("n({0}) c({1})", n, c);

    EjemploMetodos u = new EjemploMetodos();
    EjemploMetodos v = new EjemploMetodos();
    Console.WriteLine("u+v="+(u+v));
  }
}
```

2.4.4. Herencia y polimorfismo

El esquema de herencia entre clases es el mismo que se adoptó en Java: no se permite la herencia múltiple, pero existe el concepto de interfaz, el cual sí que permite la herencia múltiple. De esta forma, no existe el problema de escoger entre distintas implementaciones de un mismo método, ya que las interfaces sólo contienen los prototipos de los métodos.

La palabra reservada para referirse a la clase madre es `base`, que puede emplearse directamente en el prototipo del constructor al igual que en C++.

También es posible definir el destructor, que será llamado por el `Garbage-Collector` en el momento en el que decida eliminar el objeto. Si queremos tener control sobre este instante, deberemos usar la palabra reservada `using`, que denota un bloque de sentencias en el que los objetos declarados en su interior seran destruidos en el momento en que termine la ejecución del bloque.

Tal y como se comentaba en la definición de métodos, es posible especificarles modificadores para controlar su comportamiento en el momento en que se crea una clase derivada:

- `abstract`: método sin implementación, la cual deberá hacerse en las clases derivadas.

- `virtual`: la clase madre indica que puede ser sobrescrito por las clases derivadas.

- `override`: la clase derivada explicita que está sobrescribiendo un método de la clase madre (el cual debe estar declarado como `virtual`, `abstract` o `override`).

- `new`: la clase derivada explicita que sobrescribe un método de la clase madre aunque ésta no indicara que esto fuera posible.

```
using System;

public class EjemploHe {
  static void Main() {
    Hija x = new Hija("Pepa", 44,
                "Cardiologia", "Montanya");

    Console.WriteLine("Objeto="+x);
    Console.WriteLine("x="+x.ToString());
  }
}
abstract class Madre {
  private string n;
  private int a;

  public Madre(string n, int a) {
    this.n = n;
```

```
    this.a = a;
  }

  public new virtual string ToString() {
    return "nombre("+n+") años("+a+")";
  }
}

class Hija : Madre, Doctora, Ciclista {
  private string e, t;

  public Hija(string n, int a, string e, string t)
         :base(n, a/2) {
    this.e = e;
    this.t = t;
    Console.WriteLine("Mensaje de la madre : "+base.ToString());
  }
  public string Especialidad() { return e; }
  public string Tipus()        { return t; }
  public override string ToString() {
    return base.ToString()+" espec("+e+") tipo("+t+")";
  }
}

interface Doctora {
  string Especialidad();
}

interface Ciclista {
  string Tipus();
}
```

2.4.5. Delegados y eventos

El mecanismo para el paradigma de programación orientada a eventos de C# consiste en la existencia de clases que derivan del tipo `Event` y métodos que delegan su llamada a una colección de métodos indicados previamente. De este modo, una clase puede publicar un evento y declarar un tipo de delegado para que las clases interesadas se suscriban al evento añadiéndose a la colección de métodos del delegado. En el momento en el que se produce el evento, la clase llamará al delegado y éste se encargará de llamar consecutivamente a todos los métodos suscritos al evento.

a) `Delegate`: para delegar llamadas a otros métodos.

- Funcionalidad:
  - guarda las referencias a un conjunto de metodos;
  - cuando se invoca, se encarga de llamar a todos los métodos que tiene referenciados;
  - quien llama al delegado no conoce los métodos suscritos, simplemente le delega la tarea de llamarlos.
- Declaración:
  - se pueden declarar dentro y fuera de una clase;
  - siempre derivan de la clase `System.MulticastDelegate`;
  - sólo es posible almacenar métodos con la misma lista de parámetros y tipo de retorno.

```
using System;

delegate void Recoge (string cadena);

class AltresDelegats {
  public static Recoge recogedor;

  public static void Main () {
    recollidor = new Recoge(Tejado.RecogeEstatico);

    Balcon b = new Balcon();
    // recogedor += new Recoge(b.RecogeNoValido);

    Suelo t = new Suelo();
    recollidor += new Recoge (t.RecogePublic);

    recogedor("agua");
  }
}

class Tejado {
  public static void RecogeEstatico (string cad) {
    Console.WriteLine ("Tejado.RecogeEstatico: "+cad);
  }
}

class Suelo {
  public void RecogePublic (string cad) {
    Console.WriteLine ("Suelo.RecogePublic: "+cad);
  }
}

class Balcon {
  public Balcon () {
    AltresDelegats.recollidor += new Recoge(RecogePrivado);
  }
  void RecogePrivado (string cad) {
    Console.WriteLine ("Balcon.RecogePrivado: "+cad);
  }
  public void RecogeNoValido (int entero) {
    Console.WriteLine ("Balcon.RecogeNoValido: "+entero);
  }
}
```

b) `Event`: mecanismo de ejecución de métodos que se han suscrito a un evento en concreto.

- Los puede lanzar el mismo código o el entorno.
- Normalmente, la clase generadora no lanza el evento si no hay ningún método suscrito.
- La clase que lo publica contiene:
  - un delegado tipo `void` que recibe una referencia al objeto que genera el evento y los datos asociados,
  - un evento del tipo delegado declarado anteriormente,
  - la clase que agrupa los datos asociados al evento,
  - el código que genera el evento.
- La clase que suscribe el evento contiene:
  - el método que se suscribirá al evento,
  - el código de suscripción al evento mediante la creación de una instancia del delegado.

```
using System;
using System.Collections;
using System.Threading;

public class EventosSatelite {
  static void Main () {
    Receptor receptor = new Receptor();
    Satelite satellit = new Satelite(receptor);
    receptor.EnMarcha();
  }
}

public class Receptor {
  public delegate void InterceptaEventoInfoRecibida(
                       object s, DatosEvento args);
  public event InterceptaEventoInfoRecibida EventoInfoRecibida;

  public void EnMarcha() {
    Console.WriteLine("Receptor: en marcha esperará 2 "+
                      "señales del usuario...");
    for (int i=0; i < 2; i++) {
      Console.WriteLine("Receptor: espero usuario entre "+
                        "senyal...");
      string info  = Console.ReadLine();
      string tiempo = DateTime.Now.ToString();
     string datos = "[Rebcibida info("+info+") hora("+tiempo+")]";
      Console.WriteLine("Receptor: genero evento info_recibida "+
                        "datos="+ datos);
      if (EventoInfoRecibida != null) // si hay alguien suscrito
        EventoInfoRecibida(this, new DatosEvento(datos));
    }
  }
  public class DatosEvento : EventArgs {
    string datos;
    public string Datos { get { return datos; }
                          set { datos= value; }}
    public DatosEvento(string d){ datos = d; }
  }
}

public class Satelite {
  private Receptor receptor;

  public Satelite (Receptor receptor) {
    this.receptor=receptor;

    Console.WriteLine("Sat: nuevo");
    Console.WriteLine("Sat: suscribiendo mi "+
                     "interceptador del evento "+
                      "info_recibida del Receptor");
    receptor.EventoInfoRecibida +=
          new Receptor.InterceptaEventoInfoRecibida(
                               SatProcesaSeñal
              );
  }

  private void SatProcesaSeñal(object src,
                       Receptor.DatosEvento args) {
    Console.WriteLine("SatProcesaSeñal: "+
                      "de \"{0}\" "+
                      "datos={1}", src, args.Datos);
  }
}
```

2.4.6. *Structs, enums* y *namespaces*

En C# se han recuperado las estructuras de C++ que se habían eliminado en Java. De hecho, en C# las estructuras pueden usarse como en C++, pero también pueden incluir métodos. En efecto, una estructura en C# es como una clase pero de tipo valor. Por ejemplo, si asignamos dos variables de tipo `struct` el contenido de estas se duplica en lugar de hacer referencia al mismo objeto. El resultado es que podemos recuperar la funcionalidad y simplicidad de las estructuras de C++, e incluso les podemos añadir código para manipular su contenido. A diferencia de las clases, no será posible heredar de otras estructuras, aunque sí de interfaces.

Del mismo modo, también se han recuperado los tipos enumerados de C++ que se habían descartado en Java. Esto es muy útil para definir conjuntos de constantes si no nos preocupa su valor concreto.

Por último, hay que mencionar la existencia de los espacios de nombres bajo los cuales se pueden agrupar definiciones de clases, estructuras y enumerados, tal y como se hacia en C++ o con los paquetes de Java. La palabra reservada `using` nos sirve para ahorrarnos el prefijo de los *namespaces* en los elementos de las bibliotecas que deseemos utilizar.

```
using System;
using Mio;

public class ClassesAltres {

  static void Main() {
    // Enums
    FRUTAS f = FRUTAS.Pera;
    Console.WriteLine("f="+f);
    Array frutas = Enum.GetValues(typeof(FRUTAS));
    foreach (FRUTAS fruta in frutas)
      Console.WriteLine("FRUTAS({1})={0}", fruta,
fruta.ToString("d"));

    // Structs
    Strct s1 = new Strct(1, 2);
    Strct s2 = s1;
    Suyo.Objct o1 = new Suyo.Objct(1, 2);
    Suyo.Objct o2 = o1;

    Console.WriteLine("s1{0} s2{1}", s1, s2);
    s2.x = 5;
    Console.WriteLine("s1{0} s2{1}", s1, s2);

    Console.WriteLine("o1{0} o2{1}", o1, o2);
    o2.x = 5;
    Console.WriteLine("o1{0} o2{1}", o1, o2);
  }
}

public enum FRUTAS: byte {Manzana, Pera, Platano}

namespace Mio {
  struct Strct {
    public int x;
```

```
    public int y;
    public Strct(int x, int y) {
      this.x = x;
      this.y = y;
    }
    public override string ToString(){
      return "("+x+", "+y+")";
    }
  }
}
namespace Suyo {
  class Objct {
    public int x;
    public int y;
    public Objct(int x, int y) {
      this.x = x;
      this.y = y;
    }
    public override string ToString(){
      return "("+x+", "+y+")";
    }
  }
}
```

2.5. Boo

Boo es un lenguaje de guiones con una sintaxis parecida a la de Python pero integrado en el Common Language Interface.

Desde los archivos de código en boo, se pueden compilar a una biblioteca o ejecutable de Mono o ejecutar directamente estos archivos con el interprete del lenguaje. La principal ventaja es tener un sistema de *scripting* enlazado directamente con las bibliotecas existentes.

Web recomendada

Hay mucha documentación y ejemplos sobre el lenguaje boo en la página web del proyecto: <http://boo.codehaus.org>.

2.5.1. Estructura básica de boo

Este lenguaje nos permite realizar guiones para administración de sistemas o para pequeños programas de una manera muy sencilla. Sin tener que definir variables, funciones principales, con una gran versatilidad y con toda la potencia de las bibliotecas GTK, Ldap, Remoting, etc. de Mono.

Hay tres ejecutables:

- booc: compilador a CIL del lenguaje para poder vincular posteriormente desde otro programa en C#.
- booi: intérprete de archivos boo.
- booish: intérprete interactivo de boo.

Comparando C# y boo

Una práctica divertida es compilar un programa que haga lo mismo (un *Hola Mundo*) en boo y en C# y comprobar con monodis que el resultado obtenido en CIL es casi el mismo. Esto nos demuestra que la velocidad de un programa en C# y en boo no variará mucho si este último se compila.

2.5.2. Diferencias con C#

Boo es un lenguaje con las mismas capacidades y forma de programación que C#. Casi todo lo que podamos hacer con C# lo podremos programar en boo con las siguientes características:

- Boo no tiene definición de tipo de variables y crea una matriz si es necesario.

```
Para escribir:
  Class c = new Class (parametros);
  Type[] l = new Type[] { expr_1, expr_1,..., expr_n }
se puede codificar como:
  c = Class(parametros)
  l = (expr_1, expr_1,..., expr_n)
```

- No se usa el punto y coma.

- Usar los condicionales al final de la acción:

```
Para escribir:
   if (cond) respuesta = 42;
 se puede codificar como:
   if cond:
      respuesta = 42
 o se puede usar:
   respuesta = 42 if cond
```

- El uso de `except` en lugar de `catch` en las excepciones:

```
Para escribir:
   try { foo (); bar (); }
   catch (Exception e) { baz (); }
   finally { qux (); }
 se puede codificar como:
   try:
     foo()
     bar()
   except e:
     baz()
   ensure:
     qux()
```

- Uso de `raise` en lugar de `throw` al lanzar excepciones.

- Para hacer una conversión de una variable:

```
Se puede codificar como:
   t = cast(tipo, expr)
 o se puede usar:
   t = expr as tipo
```

- Para poder definir un método se usa `def` igual que en Python.

```
Se puede codificar como:
   static def funcion(x as int, y as string) as int:
```

- Para el constructor se usa una función llamada `constructor` y para la herencia se especifica la clase padre entre paréntesis.

```
Se puede codificar como:
   class Foo (Bar):
     def constructor(x as int):
       super(x)
       ...
```

Hay algunas diferencias más, como el posible uso de # para poner comentarios, el uso de tres comillas (“““) para definir cadenas de carácteres largas, el hecho de poder deshabilitar el control del desbordamiento de *buffer* y la opción de inicializar distintas variables en su definición.

Lo más importante es entender que es un lenguaje muy fácil de usar, sin necesidad de precompilar ni definir variables. Boo es muy útil en entornos de administración de sistemas o en *scripts* de gestión de aplicaciones.

2.5.3. Ejemplos

La estructura de un archivo boo es ésta:

- documentación del módulo,
- declaración del espacio de nombres que deben usarse,
- importación de los módulos que deben usarse,
- miembros del módulo: clases/enum/funciones,
- código principal de ejecución del *script*,
- propiedades del *assembly*.

"Hola mundo" en boo

Para ver su estructura, aquí hay un ejemplo comentado de un programa hecho en boo que escriba "Hola mundo" en consola, usando Windows Forms y con GTK:

- Usando consola:

```
print("Hola mundo!")
```

- Usando Windows Forms:

```
import System.Windows.Forms

f = Form(Text: "Hola mundo!")
f.Controls.Add(Button(Text: "Pulsarme!", Dock: DockStyle.Fill))

Application.Run(f)
```

- Usando GTK:

```
import Gtk

Application.Init()

window = Window("Hola Mundo!",
              DefaultWidth:  200,
              DefaultHeight: 150)

# La aplicación debe cerrarse cuando la ventana se cierre
window.DeleteEvent += def():
  Application.Quit ()

window.Add(Button("Pulsarme!"))
window.ShowAll()
Application.Run()
```

2.6. Nemerle

Nemerle fue creado en la Universidad de Wroclaw (Polonia) como lenguaje con definición de tipos en tiempo de compilación que ofrece la posibilidad de trabajar con los paradigmas de programación de orientación a objeto, programación funcional y programación imperativa.

Web recomendada

<http://nemerle.org>

El entorno de ejecución de Nemerle es el propio CLR, tanto la implementación Mono como la .NET. Pero en la mayoría de los casos el compilador debe obtenerse por separado. Se trata del ejecutable ncc, que se usará para compilar los archivos de código fuente en Nemerle, normalmente con extensión '.n'.

En el siguiente subapartado se repasarán varios aspectos básicos del lenguaje, y posteriormente se tratarán determinadas características que requieren una especial atención.

2.6.1. Diferencias básicas con C#

Para introducirnos rápidamente en el lenguaje Nemerle, primero haremos un repaso a las diferéncias básicas respeto al lenguaje C# expuesto anteriormente:

- Al declarar la variables y las funciones, el tipo de datos se escribe al final de la declaración:

```
funcion (x : int, y : string) : int
```

- La inferencia de tipos permite no explicitarlos, siempre que queden definidos sin ambigüedad implícitamente:

```
def x = 3
funcion (y) { y++ }
System.Console.WriteLine ("resultado"+funcion(x));
```

- Pueden hacerse definiciones de variables que no podrán cambiar su contenido una vez inicializadas (palabra reservada `def`), o bien trabajar con variables convencionales usando la palabra reservada mutable:

```
def x = 3;
mutable x = 3;
```

- No existe la palabra reservada `new`: `def var = Objeto(3);`

- No existe la palabra reservada `return`, el resultado de la última expresión de un método es el valor retornado. Esto implica que no puede interrumpirse el flujo del código de un método, ya que no puede introducirse ningún `return` en medio de éste.

- Aparte de la construcción condicional `if` habitual, también incorpora otras notaciones al estilo de Perl: `when` y `unless`.

– `when (x < 3) { ... }`: es como un `if` pero no puede tener cláusula `else`.

– `unless (x < 3) { ... }`: es como la cláusula `else` de un `if` pero sin cláusula `then`.

- Un `module` es una clase en la que todos los miembros son estáticos.

- El constructor de una clase o módulo siempre se nombra con la palabra reservada `this`, y la llamada al constructor de la clase madre se hace explícitamente en el cuerpo con la palabra reservada `base`:

```
class Clase {
  public this (x : int)
  { ... base(x); ... }
}
```

- Se puede alterar el orden de los parámetros si se indica su nombre al hacer la llamada al método:

```
funcion (edad : int, nombre : string, apellidos : string) : int {
... }

Main () : void
{
  def res1 = funcion( 20, "Ramón", "Martínez Soria");
  def res2 = funcion( apellidos = "De la Fuente", edad = 25, nombre
= "Marta");
}
```

2.6.2. Uso de funciones locales

Después de haber visto las diferencias básicas respeto de C#, en este subapartado vamos a adentrarnos en características propias de Nemerle que permiten cambiar el estilo de programación:

- Se pueden declarar funciones locales, cuyo ámbito de validez es el bloque donde están definidas. De este modo, pueden usarse fácilmente construcciones recursivas para implementar los bucles en lugar de las instrucciones iterativas explícitas:

```
module Factorial {
   public Main (): void
   {
     def fact (x: int): int {
       if (x <= 0)
         1
       else
         x * fact(x - 1)
     }
     System.Console.WriteLine ("Factorial(20)="+ fact(20))
   }
}
```

Ejemplo completo sobre la manipulación de tablas y el uso de funciones locales:

```
class ArraysTest {
  static reverse_array (ar : array [int]) : void
  {
    def loop (left, right) {
      when (left < right) {
        def tmp = ar[left];
        ar[left] = ar[right];
        ar[right] = tmp;
        loop (left + 1, right - 1)
      }
    }
    loop (0, ar.Length - 1)
  }

  static print_array (ar : array [int]) : void
  {
    for (mutable i = 0; i < ar.Length; ++i)
      Nemerle.IO.printf ("%d\n", ar[i])
  }

  static Main () : void
  {
    def ar = array [1, 42, 3];
    print_array (ar);
    Nemerle.IO.printf ("\n");
    reverse_array (ar);
    print_array (ar);
  }
}
```

2.6.3. Macros y diseño por contrato

Una de las características más interesantes de Nemerle es la existencia de macros que permiten extender el propio lenguaje. Mediante estas macros es posible trabajar basándose en la técnica de diseño por contrato, en la que se puede indicar, antes de iniciar la implementación, qué interfaces se desean y que comportamiento han de tener.

Podemos definir macros para intervenir en el proceso de compilación, ya que se trata de funciones que ejecutará el compilador. A diferencia de las macros habituales de lenguajes como C, en Nemerle las macros no son ejecutadas por un precompilador que sustituye el código que compilará el compilador, sino por el propio compilador. De esta manera, es posible manipular directamente las estructuras que usa el propio compilador a medida que analiza el código y conseguir así extender el lenguaje:

```
macro miMacro () {
  Nemerle.IO.printf ("compile-time\n");
  <[ Nemerle.IO.printf ("run-time\n") ]>;
}
```

```
ncc -r Nemerle.Compiler.dll -t:dll mimacro.n -o mimacro.dll
```

```
module Modulo {
  public Main () : void {
    miMacro ();
  }
}
```

```
ncc -r mimacro.dll miprog.n -o miprog.exe
```

Es posible seguir el paradigma de diseño por contrato usando las macros existentes para indicar las reglas que deben respetar cada método:

- `requires`: para indicar los prerrequisitos que deben cumplirse al llegar al método.

```
class Cadena
{
  public Subcadena (idxInicio : int) : string
  requires idxInicio >= 0 && idxInicio <= this.Length
  { ... }
}

* ensures: para indicar los posrequisitos que deben cumplirse al
abandonar el método.

class Lista
{
  public Vaciar () : void
  ensures this.IsEmpty
  { ... }
}
```

- `invariant`: para indicar las invariantes que deben cumplirse durante toda la ejecución del método o existencia de un objeto.

```
class Vector [T]
invariant posicion >= 0 && posicion <= arr.Length
{
  private mutable posicion : int = 0;
  private tabla : array [T] = array [];
  ...
}
```

2.6.4. Tipo `variant` y patrones

La manipulación de tipos de datos sobre conjuntos de valores enumerados puede hacerse con los tipos de datos `variant` y la construcción `match`:

```
variant Color {
   | Rojo
   | Amarillo
   | Verde
   | Diferente {
       rojo : float;
       verde: float;
       azul : float;
     }
 }
obtenerNombreColor (color : Color) : string
{
  match (color) {
    | Color.Rojo => "rojo"
    | Color.Amarillo => "amarillo"
    | Color.Verde => "verde"
    | Color.Diferente (r, g, b) =>
      System.String.Format ("rgb({0},{1},{2})", r, g, b)
    | _ => "error"
  }
}
```

Esta combinación se puede usar para tratar referencias genéricas a objetos en programación de GUI:

```
using System.Windows.Forms;

...

match (control) {
  | button is Button => ...
  | listv is ListView => ...
  | _ => ... // null case
}
```

2.6.5. Ejemplo de aplicación con GUI

En este ejemplo final podemos observar varias de las características citadas anteriormente usadas para crear un editor simple capaz de guardar y recuperar archivos de texto.

```
using System;
using Gtk;

class NMenuItem : MenuItem
{
        public name : string;
```

```
        public this(l : string)
        {
                base(l);
                name = l;
        }

        public this(l : string, e : object * EventArgs -> void)
        {
                base(l);
                name = l;
                this.Activated += e;
        }


        // This property allows us to set submenus of this menu item
        public SubmenuList : list [NMenuItem]
        {
                set
                {
                        def sub = Menu();
                        foreach (submenu in value) sub.Append (submenu);
                        this.Submenu = sub;
                }
        }
}


class MainWindow : Window
{
        /// Text input area of our window
        input : TextView;

        public this()
        {
                // Set caption of window
                base ("Very Simple Editor");
                // Resize windows to some reasonable shape
                SetSizeRequest (300,200);

                def scroll = ScrolledWindow ();
                input = TextView ();
                scroll.Add (input);

                def menu = MenuBar ();
                def mi = NMenuItem ("File");
                mi.SubmenuList =
                [
                        NMenuItem ("Open", OnMenuFile),
                        NMenuItem ("Save as...", OnMenuFile)
                ];
                menu.Append(mi);

                def vbox = VBox ();
                vbox.PackStart (menu, false, false, 0u);
                vbox.PackStart (scroll, true, true, 0u);

                // Place vertical box inside our main window
                Add (vbox);
        }

        // Handler of opening and saving files
        OnMenuFile (i : object, _ : EventArgs) : void
        {
                def mi = i :> NMenuItem;
                def fs = FileSelection (mi.name);

                when (fs.Run () == ResponseType.Ok :> int) match (mi.name)
                {
```

```
                        | "Open" =>
                                def stream = IO.StreamReader (fs.Filename);
                                input.Buffer.Text = stream.ReadToEnd();

                         | "Save as..." =>
                                def s = IO.StreamWriter(fs.Filename);
                                s.Write(input.Buffer.Text);
                                s.Close();
                         | _ => ();
                };
                fs.Hide();
        }
}

module SimpleEditor
{
        Main() : void
        {
                Application.Init();
                def win = MainWindow();

                // Exit application when editor window is deleted
                win.DeleteEvent += fun (_) { Application.Quit () };

                win.ShowAll ();
                Application.Run();
        }
}
```

3. GTK# y Gdk#

Ya hemos tratado la biblioteca GTK en el subapartado 1.2.

Para poder implementar aplicaciones gráficas en Mono hay dos opciones mayoritarias: usar Windows Forms o usar GTK. El uso de la primera biblioteca está sujeto a patentes y al control por parte de una compañía. La segunda es una biblioteca libre derivada de la aplicación GIMP y usada en el mundo del escritorio por el entorno Gnome. GTK ha mejorado mucho durante los últimos años hasta llegar a ser una de las mejores en el manejo de *widgets* para escritorio. Desde la plataforma Mono tenemos acceso a estas bibliotecas para crear nuestras aplicaciones gráficas mediante gtk-sharp. También existe gtk-dotnet.dll, que hace de puente de unión entre los *widgets* de GTK y las funciones de dibujo de Mono (`System.Drawing`).

3.1. Instalación

Para poder hacer una instalación de la última versión, lo mejor es obtener el código fuente de la página oficial de Mono. Una vez que lo tenemos, podemos descomprimirlo y compilarlo:

```
 tar -xvzf gtk-sharp-X.X.X.tar.gz
cd  gtk-sharp-X.X.X
./configure
make
make install
```

Si en el momento de la configuración nos indica que no va a compilar determinados módulos, tendremos que ir al sistema de paquetes del sistema operativo y buscar los que contengan la información de desarrollo de GTK, glade, rsvg, vte, gtkhtml, gnomevfs y gnome. Cuando el sistema detecte que están instaladas, al reconfigurar las podremos recompilar. Una vez que hemos instalado la biblioteca GTK, podremos empezar a compilar programas usando gtk-sharp.

Para poder compilar un programa usando las bibliotecas de gtk-sharp, tenemos que incluir en la línea de comandos la opción `-pkg:gtk-sharp`.

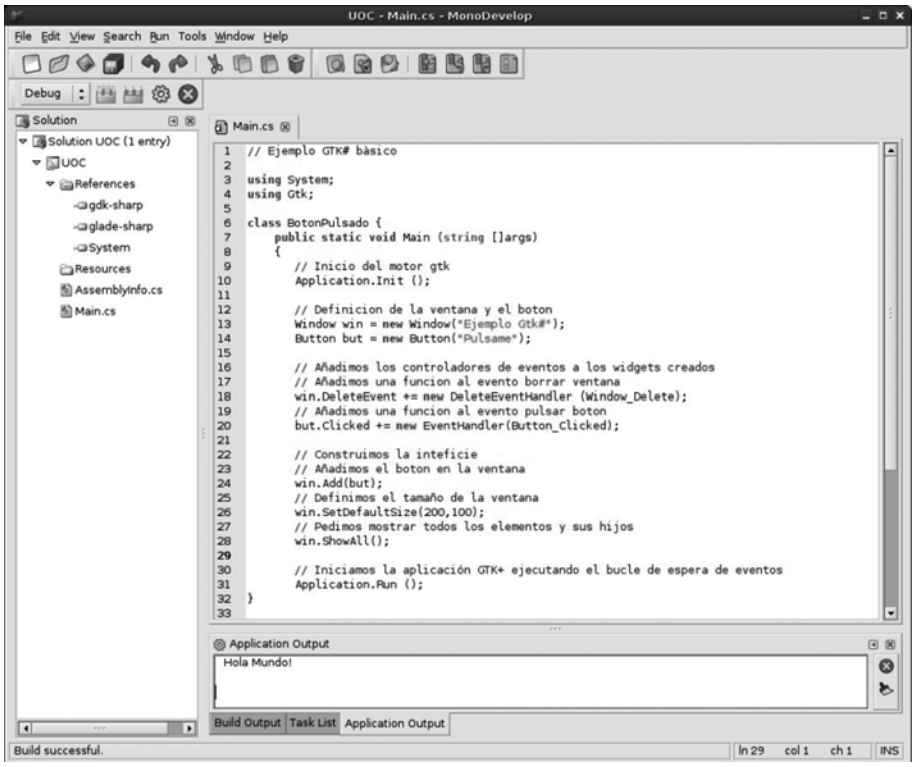
3.1.1. Ejemplo GTK#

Con este ejemplo vamos a construir un programa que nos va a mostrar un botón. Vamos a tener que usar gtk-sharp para dibujar la ventana, dibujar el botón y para usar un capturador de eventos para saber cuándo se pulsa éste.

```
// Ejemplo Gtk# básico

using System;
using Gtk;

class BotonPulsado {
public static void Main (string []args)
{
   // Inicio del motor Gtk
   Application.Init ();

   // Definición de la ventana y el botón
   Window win = new Window("Ejemplo Gtk#");
   Button but = new Button("Pulsame");

   // Añadimos los controladores de eventos a los widgets creados
   // Añadimos una función al evento borrar ventana
   win.DeleteEvent += new DeleteEventHandler (Window_Delete);
   // Añadimos una función al evento pulsar botón
   but.Clicked += new EventHandler(Button_Clicked);

   // Construimos la inteficie
   // Añadimos el bóton en la ventana
   win.Add(but);
   // Definimos el tamaño de la ventana
   win.SetDefaultSize(200,100);
   // Pedimos mostrar todos los elementos y sus hijos
   win.ShowAll();

   // Iniciamos la aplicación Gtk+ ejecutando el bucle de espera
de eventos
   Application.Run ();
}
static void Window_Delete ( object o, DeleteEventArgs args)
{
   // Salimos al pulsar el botón de cerrar ventana
   Application.Quit ();
   args.RetVal = true;
}

static void Button_Clicked ( object o, EventArgs args)
{
   // Escribimos en la consola
   System.Console.WriteLine("Hola Mundo!");
}
} // Fin clase
```

Para probar el ejemplo, podemos usar Monodevelop creando un proyecto GTK# 2.0 (figura 1). En este proyecto, vamos a tener que comprobar que las referencias a `GTK` y `system` estén bien puestas.

Figura 1. Captura de pantalla de Monodevelop con el ejemplo y las referencias



3.2. Widgets GTK#

En el subapartado anterior hemos visto un ejemplo de algunos de los *widgets* de GTK+ en uso: `Window` y `Button`. Antes de continuar con estos elementos visuales, vamos a tener que introducir el uso de los cajones (*boxes*) para el diseño de la interfaz. Ya se ha hablado de estos elementos en el subapartado de GTK+, pero ahora vamos ver un ejemplo y qué podemos hacer con estas cajas.

Aquí tenemos un árbol donde podemos observar la relación, desde el punto de vista de los objetos, de los distintos *widgets*:

```
Gtk.Object
  Gtk.Widget
    Gtk.Misc
      Gtk.Label
        Gtk.AccelLabel
      Gtk.Arrow
      Gtk.Image
    Gtk.Container
      Gtk.Bin
        Gtk.Alignment
        Gtk.Frame
        Gtk.AspectFrame
        Gtk.Button
          Gtk.ToggleButton
            Gtk.CheckButton
              Gtk.RadioButton
          Gtk.OptionMenu
```

```
        Gtk.Item
          Gtk.MenuItem
            Gtk.CheckMenuItem
              Gtk.RadioMenuItem
            Gtk.ImageMenuItem
            Gtk.SeparatorMenuItem
            Gtk.TearoffMenuItem
        Gtk.Window
          Gtk.Dialog
            Gtk.ColorSelectionDialog
            Gtk.FileSelection
            Gtk.FontSelectionDialog
            Gtk.InputDialog
            Gtk.MessageDialog
          Gtk.Plug
        Gtk.EventBox
        Gtk.HandleBox
```

```
    Gtk.ScrolledWindow
    Gtk.Viewport
  Gtk.Box
    Gtk.ButtonBox
      Gtk.HButtonBox
      Gtk.VButtonBox
    Gtk.VBox
      Gtk.ColorSelection
      Gtk.FontSelection
      Gtk.GammaCurve
    Gtk.HBox
      Gtk.Combo
      Gtk.Statusbar
Gtk.Fixed
Gtk.Paned
    Gtk.HPaned
    Gtk.VPaned
  Gtk.Layout
  Gtk.MenuShell
    Gtk.MenuBar
    Gtk.Menu
  Gtk.Notebook
  Gtk.Socket
  Gtk.Table
  Gtk.TextView
  Gtk.Toolbar
  Gtk.TreeView
Gtk.Calendar
Gtk.DrawingArea
```

```
    Gtk.Curve
  Gtk.Editable
    Gtk.Entry
      Gtk.SpinButton
  Gtk.Ruler
    Gtk.HRuler
    Gtk.VRuler
  Gtk.Range
    Gtk.Scale
      Gtk.HScale
      Gtk.VScale
    Gtk.Scrollbar
      Gtk.HScrollbar
      Gtk.VScrollbar
  Gtk.Separator
    Gtk.HSeparator
    Gtk.VSeparator
  Gtk.Invisible
  Gtk.Preview
  Gtk.ProgressBar
Gtk.Adjustment
Gtk.CellRenderer
  Gtk.CellRendererPixbuf
  Gtk.CellRendererText
  Gtk.CellRendererToggle
Gtk.ItemFactory
Gtk.Tooltips
Gtk.TreeViewColumn
```

3.2.1. Elementos contenedores

Para poder diseñar el conjunto de elementos gráficos visibles, vamos a tener que usar los distintos contenedores que hay disponibles en GTK#. Éstos son parecidos a los de la biblioteca en C.

Cajas

Estos elementos nos permiten definir la estructura en la que dividiremos las distintas ventanas permitiendo escalar fácilmente e internacionalizar la aplicación teniendo en cuenta la posibilidad de distintas cantidades de texto en los *labels* y otros *widgets*.

Hay dos tipos de cajas: `Hbox` y `Vbox`. Representan la división horizontal y vertical respectivamente. Este sistema nos permite ir dividiendo en tantas secciones como queramos hasta obtener las cajas necesarias. Para poder usar estos elementos, debemos llamar una función que dinámicamente adaptará las distintas cajas a la cantidad de texto y al tamaño de la ventana. Las dos funciones son `PackStart` y `PackEnd`. En el caso de una `Vbox`, `PackStart` empezará a empaquetar por arriba y `PackEnd` empezará por debajo. En el caso de una `Hbox`, `PackStart` empaquetará de izquierda a derecha y `PackEnd` de derecha a izquierda. Estas funciones redimensionaran el `Hbox` y `Vbox` para poder visualizar correctamente la ventana. Estas

dos funciones, igual que Vbox y Hbox, tienen algunas opciones interesantes que comentar:

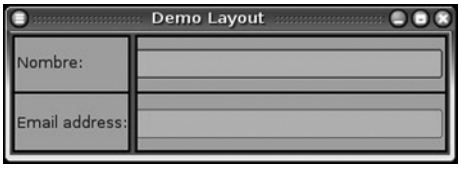
- Vbox y Hbox:
  - Spacing: número de píxeles que hay entre las distintas cajas.
  - BorderWidth: número de píxeles que hay entre el borde de la caja y el contenido.
- PackStart y PackEnd:
  - Expand: se expande para usar todo el tamaño posible.
  - Fill: hace que el *widget* ocupe todo el tamaño obtenido por expand; en caso contrario, el espacio extra será para el padding.
  - Padding: añade espacio alrededor del *widget*.

Todos los elementos gráficos que pueden ser alineados de distintas maneras implementan la interfaz Gtk.Misc. Ésta tiene las propiedades Xalign y Yalign, que toman valores de 0 (izquierda y arriba) a 1 (derecha y abajo). El uso de la función add en una caja usa internamente PackStart con los valores por defecto: true, false y 0.

```
Ejemplo de Packaging

namespace HVoxTutorial {

   using Gtk;
   using System;

   public class MainClass
   {
     public static void Main (string[] args)
     {
       Application.Init();
       SetUpGui();
       Application.Run();
     }

     static void SetUpGui()
     {
       // Creamos una ventana
       Window w = new Window("Demo Layout");

       // Creamos primero una caja horizontal y la añadimos a la
ventana
       HBox h = new HBox();
       h.BorderWidth = 6;
       h.Spacing = 6;
       w.Add(h);


       // Creamos una caja vertical
       VBox v = new VBox();
       v.Spacing = 6;

       // Añadimos la VBox en la HBox sin expand ni fill
       h.PackStart(v,false,false,0);


       // Creamos una etiqueta y la alineamos a la izquierda
       Label l = new Label ("Nombre:");
       l.Xalign=0;
```

```
        // Añadimos el label a la VBox con expand.
        v.PackStart(l,true,false,0);

        // Creamos una etiqueta y la alineamos a la izquierda
        l = new Label ("Email address:");
        l.Xalign = 0;
        v.PackStart(l,true,false,0);

        // Creamos una nueva caja vertical para los entry
        v = new VBox();
        v.Spacing = 6;
        h.PackStart(v,true,true,0);

        v.PackStart(new Entry(), true, true, 0);
        v.PackStart(new Entry(), true, true, 0);

        w.DeleteEvent += Window_Delete;
        w.ShowAll();
      }

      static void Window_Delete (object o, DeleteEventArgs args)
      {
        Application.Quit();
        args.RetVal = true;
      }
    }
}
```

Figura 2



Tablas

Para crear algunos interfaces gráficos, hace falta una estructura cuadriculada para representar los elementos gráficos. Para este caso hay un contenedor llamado "tabla". En su creación, se le puede definir:

- `Rows` y `columns`: filas y columnas que va a tener la tabla.

- `Homogenous`: booleano que nos define si todos los elementos de la tabla van a tener el mismo tamaño de altura y anchura o van a adaptar el máximo de la fila y/o columna correspondiente.

Para poder añadir un elemento gráfico en una casilla de la tabla, vamos a tener que usar las funciones `attach`: con los siguientes parámetros:

- `Widget child`: elemento gráfico que usará para poner en la celda escogida.

- `leftAttach` y `rightAttach`: columnas en las que se va a colocar el *widget*. Con estos parámetros podemos ocupar más de una celda con un elemento.

- `topAttach` y `bottomAttach`: filas en las que se va a colocar el *widget*. Con estos parámetros podemos ocupar más de una celda con un elemento.

- `xOptions`, `yOptions`:

– `Gtk.AttachOptions.Fill`**:** si la celda de la tabla es más larga que el *widget* y `Fill` está seleccionado, el *widget* se expandirá para usar todo el espacio disponible.

– `Gtk.AttachOptions.Shrink`**:** si el *widget* tiene menos espacio que el que requiere (normalmente porque el usuario ha redimensionado la ventana), luego los *widgets* normalmente desaparecerían de la ventana. Si `Shrink` es especificado, los *widgets* van a encoger con la tabla.

– `Gtk.AttachOptions.Expand`: esta opción va ha expandir la tabla hasta ocupar todo el espacio disponible en la ventana.

- `xPadding`, `yPadding`: *Padding* en las dos dimensiones.

En un caso en el que las `xOptions`, `yOptions` y `xPadding`, `yPadding` toman los valores por defecto podemos usar la función `attachdefaults` donde no hace falta mencionarlos.

Para poder definir el espacio que hay entre las distintas celdas de la tabla hay las siguientes opciones:

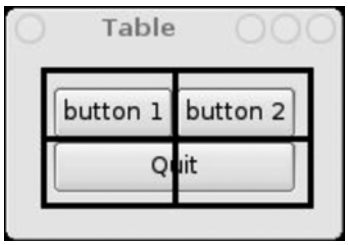
- `SetRowSpacing(int row, int spacing)` y `SetColSpacing(int column, int spacing)`: donde definimos la columna o la fila y el espaciado que queremos usar.

- `SetRowSpacing(int spacing)` y `SetColSpacings(int spacing)`: definimos el espaciado para todas las columnas y filas.

```
namespace TablasTutorial {

    using Gtk;
    using System;
    using System.Drawing;

    public class table
    {

       static void delete_event (object obj, DeleteEventArgs args)
       {
            Application.Quit();
       }

       static void exit_event (object obj, EventArgs args)
       {
            Application.Quit();
```

```
        }

        public static void Main(string[] args)
        {
             Application.Init ();
             Window window = new Window ("Table");
             window.DeleteEvent += delete_event;
             window.BorderWidth= 20;

             /* Creamos una table de 2 por 2 */
             Table table = new Table (2, 2, true);

             /* Añadimos la tabla a la ventana */
             window.Add(table);

             /* Creamos un boton */
             Button button = new Button("button 1");

             /* Ponemos el boton en la posicion de arriba a la izquierda */
             table.Attach(button, 0, 1, 0, 1);
             button.Show();

             /* Creamos otro boton */
             Button button2 = new Button("button 2");

             /* Ponemos el boton en la posicion de arriba a la derecha */
             table.Attach(button2, 1, 2, 0, 1);
             button2.Show();

             /* Creamos el boton para salir */
             Button quitbutton = new Button("Quit");

             /* Añadimos el evento para salir al boton */
             quitbutton.Clicked += exit_event;

             /* Ponemos el boton ocupando los dos espacios inferiores */
             table.Attach(quitbutton, 0, 2, 1, 2);
             quitbutton.Show();

             table.Show();
             window.ShowAll();

             Application.Run();
        }
     }
}
```

Figura 3



Frame

Herramienta que nos permite tener un contenedor con un marco alrededor. Dentro podemos poner cualquier otro contenedor o *widget*. Podemos poner un título en el marco y alinearlo donde queramos.

Constructor:

```
Frame frame1 = new Frame("Label");
```

Podemos definir distintos tipos de marcos con la propiedad `ShadowType`: `Gtk.Shadow.None`, `Gtk.Shadow.In`, `Gtk.Shadow.Out`, `Gtk.Shadow.EtchedIn` (por defecto) o `Gtk.Shadow.EtchedOut`.

```
namespace Frame {
        using Gtk;
        using System;
        using System.Drawing;


        public class frame
        {
            static void delete_event (object obj, DeleteEventArgs args)
                {
                Application.Quit();
                }

                public static void Main( string[] args)
                {

                        /* Inicializar Gtk */
                        Application.Init();

                        /* Creamos una nueva ventana */
                       Window window = new Window ("Ejemplo Frame");

                        /* Evento para cerrar la ventana */
                        window.DeleteEvent += delete_event;

                        window.SetSizeRequest(300, 300);
                        window.BorderWidth= 10;

                        /* Creamos un frame */
                        Frame frame = new Frame("Frame");
                        window.Add(frame);

                        /* Añadimos un título */
                        frame.Label = "Nuestro frame";

                        /* Decidimos el tipo de marco */
                        frame.ShadowType = (ShadowType) 4;
                        frame.Show();

                        window.ShowAll();
                        Application.Run();

                }
        }
}
```

Contenido fijo

En este contenedor vamos a poder poner los *widgets* donde queramos de una manera fija. La gran ventaja es la facilidad para ver cómo quedará. La desventaja es que al crecer no se redimensiona.

Las funciones usadas son:

```
Fixed fixed1 = new Fixed(); - Constructor
fixed1.Put(widget, x, y); - Para poder poner un elemento gráfico
en una posición.
fixed1.Move(widget, x, y); - Para mover un elemento gráfico de una
posición a otra.
```

Layout

Este contenedor es igual que el anterior con la diferencia de tener una barra desplazadora que nos permite disponer de un espacio casi infinito donde colocar los *widgets*. Este contenedor nos permite eludir el límite de 32.767 píxeles del presentado anteriormente. Las funciones son las mismas excepto que en el constructor definimos el alineamiento y aparece una propiedad para definir el tamaño del *layout*.

```
layout1.Size = new Size (width, height)
```

3.2.2. Elementos gráficos

Botones

Podemos distinguir los siguientes tipos de botones:

1) Botones normales

Son los botones en los que tenemos un pulsador para activar un evento. Este tipo de *widgets* se pueden crear de dos formas:

```
Button button1 = new Button("texto")
Button button2 = new ButtonWithMnemonic()
```

Al tener un botón sin ningún texto, se obtiene dicho botón con una caja contenedora donde podemos añadir otros tipos de *widgets*, como imágenes y *labels*.

```
namespace Botones1 {

        using Gtk;
        using System;

        public class boton
       {
```

```
 /* Función del signal de pulsar el botón */
                static void callback( object obj, EventArgs args)
                {
                       Console.WriteLine("Hola, el boton ha sido pulsado");
                }

                /* another callback */
               static void delete_event (object obj, DeleteEventArgs args)
                {
                        Application.Quit();
                }

                public static void Main(string[] args)
                {
                        Application.Init();

                        /* Creamos una ventana */
                        Window window = new Window ("Botones");
                        window.DeleteEvent += delete_event;
                        window.BorderWidth = 10;

                        /* Creamos un boton */
                        Button button = new Button();

                        /* Añadimos una señal de callback */
                        button.Clicked += callback;

                        /* Cremos una caja para la etiqueta */
                        HBox box = new HBox(false, 0);
                        box.BorderWidth =  2;

                        /* Creamos una etiqueta para el boton */
                        Label label = new Label ("Pulsar!");
                        box.PackStart(label, false, false, 3);
                        label.Show();

                        /* Muestra la caja */
                        box.Show();

                        /* Añade la caja al boton */
                        button.Add(box);

                        /* Muestra el boton */
                        button.Show();

                        window.Add(button);
                        window.ShowAll();
                        Application.Run();
                }
        }
}
```

2) Botones de estado

La diferencia entre este botón y el anterior es que guarda el estado en una propiedad llamada `Active`. Si pulsamos el botón, éste se queda pulsado y la propiedad obtiene el valor de `true`. En caso contrario, pasa a `false`.

```
using Gtk;
using System;

public class Botones2
{
```

```
     public static void Main(string[] args)
     {
          Application.Init();
          Window window = new Window("Bontones");
          window.DeleteEvent += delete_event;

          /* Creamos un boton de estado */
          ToggleButton togglebutton = new ToggleButton("boton");
          togglebutton.Clicked += clickedCallback;

          window.Add(togglebutton);
          window.ShowAll();

          Application.Run();
      }

      static void delete_event (object obj, DeleteEventArgs args)
      {
          Application.Quit();
      }

      static void clickedCallback (object obj, EventArgs args)
      {
          /* Para saber si está activo */
          if (((ToggleButton) obj).Active)
               Console.WriteLine ("Estoi Activo");
      }
}
```

3) Botones de chequeo

Es lo mismo que los botones anteriores, con la diferencia de que se renderiza con un pequeño recuadro con texto al lado. Este recuadro queda marcado cuando se activa. Las funciones de creación son:

```
GtkWidget CheckButton1 = new CheckButton();
GtkWidget CheckButton1 = new CheckButtonWithLabel(string label);
GtkWidget CheckButton1 = new CheckButtonWithMnemonic(string label);
```

4) Botones de radio

Estos elementos sirven para tener un conjunto de chequeo que son autoexclusivos. Cuando escogéis un elemento de éstos, se desactivan los otros que están en el mismo grupo. Para crear un botón en un grupo, modificad la propiedad de `Group`.

```
namespace Botons {

   using Gtk;
   using System;

   public class radiobuttons
   {
```

```
      static void delete_event (object obj, DeleteEventArgs args)
      {
           Application.Quit();
      }

      public static void Main(string[] args)
      {
          Application.Init();

          /* Creamos la ventana con los botones */
          Window window = new Window("Botons");
          window.DeleteEvent += delete_event;
          window.BorderWidth = 0;

          VBox box1 = new VBox (false, 0);
          window.Add(box1);
          box1.Show();

          /* Creamos un radio button */
          RadioButton radiobutton = new RadioButton  (null, "button1");
          box1.PackStart(radiobutton, true, true, 0);
          radiobutton.Show();

          /* Creamos un segondo radio button y lo ponemos en el mismo grupo */
          RadioButton radiobutton2 = new RadioButton(null, "button2");
          radiobutton2.Group=radiobutton.Group;
          /* Activamos este radio button */
          radiobutton2.Active = true;
          box1.PackStart(radiobutton2, true, true, 0);
          radiobutton2.Show();

          window.ShowAll();
          Application.Run();
       }
    }
}
```

Inputs

Para poder recoger datos, tenemos distintos tipos de *widgets*:

- **`Entry`** – Elemento de entrada de una sola línea. Podemos acceder al texto con la propiedad `Text`, decidir si es editable o no con la propiedad `iseditable` y copiar y pegar del portapapeles con la función `pasteclipboard`, `copyclipboard` y `cutclipboard`.

- **`Combobox`**, **`ComboboxEntry`** – Elementos de entrada para poder escoger entre un desplegable. En `ComboboxEntry` podemos escoger, escribiendo el valor que queremos elegir. Para poder trabajar en este último, se trata como una `Entry` normal.

```
using System;
using Gtk;

class ComboBoxSample
{
     static void Main ()
     {
```

```
            new ComboBoxSample ();
        }
        ComboBoxSample ()
        {
             Application.Init ();
             /* Creamos una ventana */
             Window win = new Window ("ComboBoxSample");
            win.DeleteEvent += new DeleteEventHandler (OnWinDelete);

             /* Creamos una combobox */
             ComboBox combo = ComboBox.NewText ();
             /* Añadimos 5 elementos con un texto */
             for (int i = 0; i < 5; i ++)
                  combo.AppendText ("item " + i);
             /* Añadimos el controlador del evento */
             combo.Changed += new EventHandler (OnComboBoxChanged);

             win.Add (combo);
             win.ShowAll ();
             Application.Run ();
        }

        void OnComboBoxChanged (object o, EventArgs args)
        {
             /* Recogemos el combo escogido */
             ComboBox combo = o as ComboBox;
             if (o == null)
                  return;

             TreeIter iter;
            /* Obtenemos el combo escogido y mostramos el mensaje */
             if (combo.GetActiveIter (out iter))
               Console.WriteLine ((string) combo.Model.GetValue (iter,
0));
        }

        void OnWinDelete (object obj, DeleteEventArgs args)
        {
             Application.Quit ();
        }
}
```

- **`TextView`** – Elemento para mostrar un cuadro de texto con una barra desplazadora. En este elemento vamos a mostrar un *textbuffer* que podemos obtener de un archivo o crearlo. Dentro del elemento, tenemos un atributo `buffer` donde podemos usar la propiedad de `Text` para leer o editar el contenido.

Scrollbars, *Range* y *Scale*

Este conjunto de *widgets* sirve para implementar elementos donde hay una lista de opciones para escoger y una barra de desplazamiento que permite al usuario escoger el valor. El *widget* básico es el de `Range`, que asociado a un elemento `Adjustments` puede calcular el tamaño del elemento de desplazamiento y la posición dentro de la barra.

Estos *widgets* tienen una política de actualización que nos permite definir cuándo cambiamos el valor asociado. El atributo `UpdateType` puede ser `Continuous`, con el que se lanza un evento `ValueChanged` cada vez que el

usuario mueve la barra desplazadora o el escalador. Otra opción es que sea `Discontinuous`, con lo que no se modifica el valor hasta que el usuario ha despulsado el *widget*. La última opción es `Delayed`, donde se cambia el valor cada vez que hace un momento que el usuario no mueve el elemento de desplazamiento dentro de la barra y lo mantiene pulsado.

```
namespace Range {
        using Gtk;
        using System;
        using System.Drawing;


        public class EjemploRange
        {
                static HScale hscale;
                static VScale vscale;

                static void scale_set_default_values (Scale s)
                {
                        s.UpdatePolicy = UpdateType.Continuous;
                        s.Digits = 1;
                        s.ValuePos = PositionType.Top;
                        s.DrawValue = true;
                }

                static void create_range_controls ()
                {
                        Window window;
                        VBox box1, box3;
                        HScrollbar scrollbar;
                        Scale scale;
                        Adjustment adj1, adj2;

                        window = new Window ("Range");
                        box1 = new VBox (false, 0);
                        window.Add (box1);
                        box1.ShowAll ();

                        /* Creamos un conjunto de ajustes para el VScale */
                        adj1 = new Adjustment (0.0, 0.0, 101.0, 0.1, 1.0, 1.0);
                        vscale = new VScale ((Adjustment) adj1);

                        /* Ponemos los valores por defecto */
                        scale_set_default_values (vscale);

                        box1.PackStart (vscale, true, true, 0);
                        vscale.ShowAll ();
                        box3 = new VBox (false, 10);
                        box1.PackStart (box3, true, true, 0);
                        box3.ShowAll ();

                        /* Utilizamos los mismos ajustes */
                        hscale = new HScale ((Adjustment) adj1);
                        hscale.SetSizeRequest (200, -1);

                        /* Ponemos los valores por defecto */
                        scale_set_default_values (hscale);

                        box3.PackStart (hscale, true, true, 0);
                        hscale.ShowAll ();

                        /* utilizamos los mismos ajustes esta vez para un scrollbar */
                        scrollbar = new HScrollbar ((Adjustment) adj1);

                        /* Aquí no ponemos los valores por defecto */
```

```
                        scrollbar.UpdatePolicy = UpdateType.Continuous;

                        box3.PackStart (scrollbar, true, true, 0);
                        scrollbar.ShowAll ();

                        /* Creamos unos nuevos ajustes */
                        adj2 = new Adjustment (1.0, 0.0, 5.0, 1.0, 1.0, 0.0);
                        scale = new HScale (adj2);
                        scale.Digits = 0;

                        box1.PackStart (scale, true, true, 0);
                        scale.ShowAll ();
                        box1.ShowAll ();

                        /* Creamos unos nuevos ajustes */
                        adj2 = new Adjustment (1.0, 1.0, 101.0, 1.0, 1.0, 0.0);
                        scale = new HScale (adj2);
                        scale.Digits = 0;
                        box1.PackStart (scale, true, true, 0);
                        scale.ShowAll ();
                        window.ShowAll ();
                }

                public static void Main (string [] args)
                {
                        Application.Init ();
                        create_range_controls ();
                        Application.Run ();
                }
        }
}
```

Progress Bars

Widget para tener una barra que va incrementando. Con el atributo `Fraction` podemos escoger en qué punto está del proceso, un valor de `0` a `1`. Podemos escoger si va de derecha a izquierda o viceversa, de arriba a abajo o al revés. También se puede usar para mostrar que hay algún tipo de actividad con la función `Pulse`. El incremento dado por un pulso se define con la propiedad `PulseStep`.

Éste es un ejemplo completo donde podemos ver el uso de temporizadores, tablas, botones y barras de desplazamiento.

```
using GLib;
using Gtk;
using System;

class ProgressBarSample {

    /* Datos que enviamos en los callbacks */
  public struct ProgressData {
  public Gtk.Window window;
    public Gtk.ProgressBar pbar;
    public uint timer;
    public bool activity_mode;
  }

  static ProgressData pdata;
```

```
  /* Función que va cambiando el valor del scrollbar */
  static bool progress_timeout()
  {
    double new_val;
    if (pdata.activity_mode)
        /* En caso de estar en un modo pulse */
      pdata.pbar.Pulse();
    else {
      /* En caso de esar en modo fracción*/
      new_val = pdata.pbar.Fraction + 0.01;
      if (new_val > 1.0)
        new_val = 0.0;

      /* Ponemos el nuevo valor */
      pdata.pbar.Fraction = new_val;
    }
    return true;
  }

  /* Cambiamos el texto que se mostrará en el scrollbar */
  static void toggle_show_text (object obj, EventArgs args)
  {
    if (pdata.pbar.Text == "")
      pdata.pbar.Text = "some text";
    else
      pdata.pbar.Text = "";
  }

  /* Cambiamos el modo de actividad de pulsos a fracción */
  static void toggle_activity_mode (object obj, EventArgs args)
  {
    pdata.activity_mode = !pdata.activity_mode;
    if (pdata.activity_mode)
      pdata.pbar.Pulse();
    else
      pdata.pbar.Fraction = 0.0;
  }

  /* Cambiamos la orientación del scrollbar */
  static void toggle_orientation (object obj, EventArgs args)
  {
    switch (pdata.pbar.Orientation) {
      case Gtk.ProgressBarOrientation.LeftToRight:
        pdata.pbar.Orientation =
Gtk.ProgressBarOrientation.RightToLeft;
        break;
      case Gtk.ProgressBarOrientation.RightToLeft:
        pdata.pbar.Orientation =
Gtk.ProgressBarOrientation.LeftToRight;
        break;
      }
  }

  static void destroy_progress (object obj, DeleteEventArgs args)
  {
    app_quit();
  }

  static void button_click (object obj, EventArgs args)
  {
    app_quit();
  }

  static void app_quit() {
      /* Paramos el temporizador */
    GLib.Source.Remove (pdata.timer);
    pdata.timer = 0;
    Application.Quit ();
```

```
  }

  static void Main()
  {
    Gtk.HSeparator separator;
    Gtk.Table table;
    Gtk.Button button;
    Gtk.CheckButton check;
    Gtk.VBox vbox;

    Application.Init ();

    /* Reservamos el espacio para los datos que enviamos a los
callbacks */
    pdata = new ProgressData();
    pdata.activity_mode = false;
    pdata.window = new Gtk.Window(Gtk.WindowType.Toplevel);
    pdata.window.Resizable = true;
    pdata.window.DeleteEvent += destroy_progress;
    pdata.window.Title = "GtkProgressBar";
    pdata.window.BorderWidth = 0;

    vbox = new Gtk.VBox(false, 5);
    vbox.BorderWidth = 10;
    pdata.window.Add(vbox);
    vbox.Show();

    /* Alineamos al centro */
    Gtk.Alignment align = new Gtk.Alignment( 1, 1, 0, 0);
    vbox.PackStart(align, false, false, 5);
    align.Show();

    /* Creamos la GtkProgressBar */
    pdata.pbar = new Gtk.ProgressBar();
    pdata.pbar.Text = "";
    align.Add(pdata.pbar);
    pdata.pbar.Show();

    /* Creamos un temporalizador para poder canviar el valor */
    pdata.timer = GLib.Timeout.Add(100, new GLib.TimeoutHandler
(progress_timeout) );


    separator = new Gtk.HSeparator();
    vbox.PackStart(separator, false, false, 0);
    separator.Show();
    table = new Gtk.Table(2, 3, false);
    vbox.PackStart(table, false, true, 0);
    table.Show();

    /* Creamos un check para escoger si vemos o no el texto */
    check = new Gtk.CheckButton("Show text");
    table.Attach(check, 0, 1, 0, 1,
        Gtk.AttachOptions.Expand | Gtk.AttachOptions.Fill,
        Gtk.AttachOptions.Expand | Gtk.AttachOptions.Fill,
        5, 5);
    check.Clicked += toggle_show_text;
    check.Show();

    /* Creamos un check para escoger el modo de actividad */
    check = new Gtk.CheckButton("Activity mode");
    table.Attach(check, 0, 1, 1, 2,
        Gtk.AttachOptions.Expand | Gtk.AttachOptions.Fill,
        Gtk.AttachOptions.Expand | Gtk.AttachOptions.Fill,
        5, 5);
    check.Clicked += toggle_activity_mode;
    check.Show();
```

```
    /* Creamos un check para escoger la orientación */
    check = new Gtk.CheckButton("Right to Left");
    table.Attach(check, 0, 1, 2, 3,
        Gtk.AttachOptions.Expand | Gtk.AttachOptions.Fill,
        Gtk.AttachOptions.Expand | Gtk.AttachOptions.Fill,
        5, 5);
    check.Clicked += toggle_orientation;
    check.Show();

    button = new Gtk.Button("close");
    button.Clicked += button_click;
    vbox.PackStart(button, false, false, 0);
    button.CanDefault = true;
    button.GrabDefault();
    button.Show();

    pdata.window.ShowAll();

    Application.Run ();
  }
}
```

Labels

Este *widget* es el usado en el caso de querer mostrar un trozo de texto no editable. Para poder tener eventos en él hace falta ponerlo dentro de un `EventBox` o un botón. Podemos escoger la justificación del texto de la siguiente forma:

```
label.Justify = Justification.Left;
```

Si queremos marcar un texto, podemos usar la propiedad de `Pattern`, donde le podemos asignar qué caracteres van subrayados, por ejemplo:

```
label.Pattern ="___ __ ____    __"
```

Tooltips

Los *tooltips* son mensajes que aparecen al mantener el cursor encima de un *widget* durante unos segundos. Usarlo es fácil: primero creamos el objeto `Tooltip`, el *widget* en el que queremos enlazarlo y se usa la siguiente función:

```
tooltip1.SetTip(widget, tooltipText, tooltipPrivate);
```

En esta función, el primer argumento es el elemento gráfico; el segundo, el texto que se mostrará; y el tercero puede ser nulo o un enlace a un elemento de ayuda contextual.

Flechas

Este *widget* nos permite crear una imagen de una flecha para poder añadirla en otros *widgets*. Podemos escoger su forma en la creación o posteriormente:

```
Arrow arrow1 = new Arrow( arrow_type, shadow_type );
arrow1.SetArrow(arrow_type, shadow_type );
```

Donde aquí podemos escoger como tipos:

- `Gtk.Arrow.Up`
- `Gtk.Arrow.Down`
- `Gtk.Arrow.Left`
- `Gtk.Arrow.Right`

Y como tipo de sombra:

- `Gtk.Shadow.In`
- `Gtk.Shadow.Out`
- `Gtk.Shadow.Etched.In`
- `Gtk.Shadow.Etched.Out`

Treeview

Éste es seguramente el *widget* más completo e importante de todos. Nos permite mostrar datos de un modo estructurado en una lista o en un árbol, filtrar, editar y mostrar iconos de esa lista. Está diseñado con un modelo Modelo Vista Controlador. Este sistema nos permite tener la información que se desea mostrar, cómo la vamos a ver y finalmente cómo la vamos a tratar.

Para poder tratar los distintos tipos de datos, hay dos tipos de objetos:

- `ListStore`: sirve para guardar información en forma de lista.

- `TreeStore`: sirve para guardar información en forma de árbol.

Para poder visualizar la información, los `Treeview` se descomponen en:

- `Treeview widget`: responsable de la visualización del *widget* y de la interacción del usuario.

- `TreeViewColumn`: responsable de una columna que contiene un `CellRenderer`.

- `CellRenderer`: elemento mínimo del *widget*. Se pueden usar separados de la columna para poder componerlos, se pueden usar varios en una co-

lumna para poner una imagen y texto juntos, por ejemplo. Éste puede ser de distintos tipos:

- `CellRendererText`: para mostrar el texto.
- `CellRendererPixbuf`: para mostrar las imágenes.
- `CellRendererProgress`: para mostrar las barras de progreso.
- `CellRendererCombo`: para mostrar una caja desplegable.
- `CellRendererToggle`: para mostrar una opción de chequeo.

En este ejemplo podemos ver cómo filtramos los datos de una lista donde añadimos la información desde el código.

```
public class EjemploLista
{
  public static void Main ()
  {
    Gtk.Application.Init ();
    new EjemploLista ();
    Gtk.Application.Run ();
  }

  Gtk.Entry filterEntry;

  Gtk.TreeModelFilter filter;

  public EjemploLista ()
  {
    // Creamos una ventana
    Gtk.Window window = new Gtk.Window ("TreeView Example");
    window.SetSizeRequest (500,200);

    // Entry para poder filtrar
    filterEntry = new Gtk.Entry ();

    // Evento para actuar en el cambio de la Entry anterior
    filterEntry.Changed += OnFilterEntryTextChanged;
    Gtk.Label filterLabel = new Gtk.Label ("Artist Search:");

    // Ponemos la caja en la parte superior
    Gtk.HBox filterBox = new Gtk.HBox ();
    filterBox.PackStart (filterLabel, false, false, 5);
    filterBox.PackStart (filterEntry, true, true, 5);

    // Creamos el TreeView
    Gtk.TreeView tree = new Gtk.TreeView ();

    // Ponemos el Tree view en una caja
    Gtk.VBox box = new Gtk.VBox ();
    box.PackStart (filterBox, false, false, 5);
    box.PackStart (tree, true, true, 5);
    window.Add (box);

    // Una columna para el artista
    Gtk.TreeViewColumn artistColumn = new Gtk.TreeViewColumn ();
    artistColumn.Title = "Artista";

    // Creamos la CellRender para mostar el nombre
    Gtk.CellRendererText artistNameCell = new Gtk.CellRendererText ();

    // Añadimos la Cellrender a la columna
    artistColumn.PackStart (artistNameCell, true);

    // Creamos una columna para la canción
```

```
    Gtk.TreeViewColumn songColumn = new Gtk.TreeViewColumn ();
    songColumn.Title = "Canción";

    // Hacemos lo mismo que con el artista
    Gtk.CellRendererText songTitleCell = new Gtk.CellRendererText ();
    songColumn.PackStart (songTitleCell, true);

    // Añadimos las columnas a la lista
    tree.AppendColumn (artistColumn);
    tree.AppendColumn (songColumn);

    // Le decimos a las columas qué información vamos a mostrar
    artistColumn.AddAttribute (artistNameCell, "text", 0);
    songColumn.AddAttribute (songTitleCell, "text", 1);

    // Creamos una lista de dos string por fila
    Gtk.ListStore musicListStore = new Gtk.ListStore (typeof (string),
                                                      typeof (string));

    // Añadimos la información
    musicListStore.AppendValues ("Danny Elfman", "This is hallowen");
    musicListStore.AppendValues ("Danny Elfman", "Chicago Soundtrack");
    musicListStore.AppendValues ("Eleftheria", "Dinata");
    musicListStore.AppendValues ("Alkistis", "Lava");

    /* En lugar de asignar el modelo directamente al Treeview, creamos un
       TreeModelFilter, que estará entre el modelo ( la ListStore) y
la visualización
       ( el TreeView ) filtrando el modelo que se visualiza.Actuaría
como
       Controlador. */
    filter = new Gtk.TreeModelFilter (musicListStore, null);

    // Definimos qué función determinará qué fila se visualizará y
cuál no
    filter.VisibleFunc = new Gtk.TreeModelFilterVisibleFunc
(FilterTree);

    /* Asignamos el modelo del TreeView como el filtro. En caso de no
usar el
    filtro, asignaríamos directamnte la lista */
    tree.Model = filter;

    // Mostralo todo
    window.ShowAll ();
  }

  private void OnFilterEntryTextChanged (object o, System.EventArgs
args)
  {
    // Llamamos al filtro para que reactúe
    filter.Refilter ();
  }

  /* Función para el filtro */
  private bool FilterTree (Gtk.TreeModel model, Gtk.TreeIter iter)
  {
    string artistName = model.GetValue (iter, 0).ToString ();

    if (filterEntry.Text == "")
      return true;

    if (artistName.IndexOf (filterEntry.Text) > -1)
      return true;
    else
      return false;
  }
}
```

Para poder crear un árbol, en lugar de una lista tenemos:

```
Gtk.TreeStore musicListStore = new Gtk.TreeStore (typeof (string),
typeof (string));
```

Cuando creamos un nuevo valor en el modelo, asignamos quién es su padre:

```
Gtk.TreeIter iter = musicListStore.AppendValues ("BSO");
musicListStore.AppendValues (iter, "Danny Elfman", "Town meeting
song");
```

Podemos assignar una función a una columna para obtener los datos. De este modo, podemos tener una estructura compleja de información, y a partir de estas funciones donde tenemos el iterador de una lista de objetos podemos decidir qué información vamos a mostrar.

Para hacer esto, primero tenemos que crear una estructura de datos para nuestro modelo:

```
public class Song
{
        public Song (string artist, string title)
        {
                this.Artist = artist;
                this.Title = title;
        }

        public string Artist;
        public string Title;
}
```

Luego decidimos qué funciones nos van a dar la información:

```
private void RenderSongTitle (Gtk.TreeViewColumn column,
Gtk.CellRenderer cell, Gtk.TreeModel model, Gtk.TreeIter iter)
{
        Song song = (Song) model.GetValue (iter, 0);
        (cell as Gtk.CellRendererText).Text = song.Title;
}
private void RenderArtistName (Gtk.TreeViewColumn column,
Gtk.CellRenderer cell, Gtk.TreeModel model, Gtk.TreeIter iter)
{
        Song song = (Song) model.GetValue (iter, 0);
        if (song.Artist.StartsWith ("X") == true) {
               (cell as Gtk.CellRendererText).Foreground = "red";
        } else {
                (cell as Gtk.CellRendererText).Foreground =
"darkgreen";
        }
        (cell as Gtk.CellRendererText).Text = song.Artist;
}
```

Finalmente, añadimos una lista de objetos `Song` y la definimos como modelo del `TreeView`. Una vez hecho esto, sólo nos falta definir las funciones como elementos para decidir el contenido de las celdas.

```
songs = new ArrayList ();
songs.Add (new Song ("Danny Elfman", "Making Christmas"));

Gtk.ListStore musicListStore = new Gtk.ListStore (typeof (Song));
foreach (Song song in songs) {
       musicListStore.AppendValues (song);
}

artistColumn.SetCellDataFunc (artistNameCell, new
Gtk.TreeCellDataFunc (RenderArtistName));
songColumn.SetCellDataFunc (songTitleCell, new
Gtk.TreeCellDataFunc (RenderSongTitle));

tree.Model = musicListStore;
```

Podemos definir que una celda es editable y asignar una función como evento de su edición:

```
artistNameCell.Editable = true;
artistNameCell.Edited += artistNameCell_Edited;
```

Un ejemplo de una función para recibir este evento es:

```
private void artistNameCell_Edited (object o, Gtk.EditedArgs args)
{
 Gtk.TreeIter iter;
 // Obtenemos el iterador donde se ha editado
 musicListStore.GetIter (out iter, new Gtk.TreePath (args.Path));

 Song song = (Song) musicListStore.GetValue (iter, 0);
 song.Artist = args.NewText;
}
```

Menús

En GTK# hay tres tipos de clases para hacer menús:

- `Gtk.MenuItem`: representa un elemento en un menú.
- `Gtk.Menu`: recoge todos los elementos en un menú y crea submenús para los ítems.
- `Gtk.MenuBar`: crea una barra con todos los ítems y crea submenús.

Para poder crear uno, vamos a tener que:

- crear un `MenuBar`,
- crear un `MenuItem` para cada elemento del primer nivel:
  - añadimos un `Menu` como submenú para cada elemento de primer nivel,

– creamos cada elemento del submenú,
– añadimos los elementos de primer nivel a `MenuBar`.

Se pueden crear claves `Mnemonic` que nos permiten acceder a éstas directamente desde el teclado con una combinación de la tecla Alt y la letra escogida.

Para empezar un ejemplo, tenemos la creación del `Menubar` y los elementos de primer nivel:

```
MenuBar mn = new MenuBar();

Menu file_menu = new Menu();
MenuItem item = new MenuItem ("_File");
item.Submenu = file_menu;
mv.Append(item);

Menu edit_menu = new Menu();
item = new MenuItem("_Edit");
item.Submenu = edit_menu;
mb.Append(item);
```

Podemos añadir elementos del `Stock` y separadores:

```
// Elemento que se encarga de gestionar todos los aceleradores de
teclado de la ventana
AccelGroup agrp = new AccelGroup();
window.AddAccelGroup(agrp);

item = new ImageMenuItem(Stock.Open,agrp);
file_menu.Append(item);
file_menu.Append(new SeparatorMenuItem ());
```

O podemos añadir elementos manuales:

```
item = new MenuItem("_Transoform");
edit_menu.Append(item);
```

También se pueden usar elementos del menú más complejos como `CheckMenuItem`, donde podemos tener un botón de chequeo o el `RadioMenuItem` para incluir una opción de escoger.

Para poder hacer accesibles desde el teclado las opciones en el caso de no ser ítems de `Stock`, vamos a tener que añadir los aceleradores a mano:

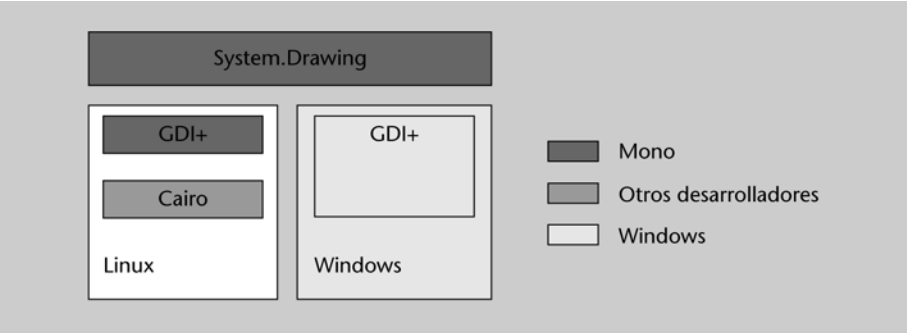
```
/* Escogemos la letra que usaremos y el modificador ( la R y el
Control ) y definimos
que la acceleración sea visible en el menú */
item.AddAccelerator ("activate",agrp,
   new AccelKey ( Gdk.key.R,
Gdk.ModifierType.ControlMask,AccelFlags.Visible));
```

3.2.3. *Canvas*

En Mono hay dos maneras de dibujar: con la API de la libgdi `System.Drawing` y con la biblioteca Cairo. Aparte de estas bibliotecas, tenemos la opción, más eficiente, de la `PixBuf` para imágenes y `Pango` para texto. Aquí vamos a explicar cómo usar estas cuatro bibliotecas.

Para entender cómo funciona la biblioteca para dibujar tenemos la biblioteca libgdi nativa en Windows y portada a Linux por el proyecto Mono. Esta biblioteca es la base de `System.Drawing`. En Linux disponemos de la biblioteca Cairo, que nos permite tener el espacio de nombres `Mono.Cairo` para el acceso a sus *bindings*. Tal y como se dijo anteriormente, GTK# está implementado sobre de la biblioteca Cairo desde la versión 2.8 a bajo nivel.

Figura 4



Dibujar

La API de `System.Drawing` es parecida a la forma que tiene PDF 1.4 para definir la estructura del documento. Es una biblioteca que implementa la misma API de libgdi pero permitiendo el acceso desde C# a ésta. En Linux, la implementación de libgdi está basada en Cairo para las operaciones gráficas de alto rendimiento.

Imágenes y DrawingArea

Para poder trabajar con áreas de images y gráficos desde GTK#, debemos usar estos dos tipos de datos:

- **`Gtk.DrawingArea`**: *widget* de GTK para definir un área de dibujo.

- **`Gdk.Pixbuf`** y **`Gdk.Pixmap`**: objetos de GDK que nos permiten trabajar con imágenes de bits. Éstos son usados en cualquier sitio donde necesitemos una imagen, sea un botón, un `TreeView` o una `Drawing Area`.

Para poder cargar una imagen de un archivo, vamos a tener que usar el objeto `Pixbuf`:

```
Gdk.Pixbuf buffer = new Pixbuf(null,"archivo.png"); # este archivo
puede ser un resource dentro o fuera del Assembly
```

Una vez que tenemos la imagen cargada en un *buffer*, podemos escalarla o trabajar con ella a nuestro gusto con un conjunto de funciones de la clase.

Para crear un área de dibujo en GTK, tenemos el constructor:

```
Gtk.DrawingArea darea = new DrawingArea();
```

Una vez que tenemos el área, podemos saber su tamaño con la propiedad `Allocation` y dibujar en ella un pixmap con la función `DrawDrawable` del Gdk de la Window de `DrawingArea`. Los pixmap nos sirven para hacer un *double-buffer* y componer en este espacio de memoria la imagen que queremos mostrar. Este objeto tiene la función `DrawPixbuf`, que nos permite mapear un pixbuf a un pixmap. Cada vez que cambiamos un pixmap, tenemos que llamar a la función `QueueDrawArea` de la `DrawingArea` para lanzar el evento de `Expose`.

```
...
// Creamos un área
darea = new DrawingArea();
darea.SetSizeRequest(200,300);

// Añadimos los eventos
darea.ExposeEvent += Expose_Event;
darea.ConfigureEvent += Configure_Event;
...

// colocar en un pixmap un pixbuf y mostrarlo en la drawing area
static voig PonerPixbuf (GdkPixbuf buf)
{
  // de la misma forma podriamos dibujar líneas, polígonos, arcos,
...
  pixmap.DrawPixbuf(darea.Style.BlackGC, buf, 0, 0, 0, 0,
buf.Width, buf.Height, RgbDither.None, 0, 0);
  darea.QueueDrawArea (0, 0, buf.Width, buf.Height);
}

// Configuramos el área de dibujo
static void Configure_Event (object obj, ConfigureEventArgs args)
  Gdk.EventConfigure ev = args.Event;
  Gdk.Window window = ev.Window;
  Gdk.Rectangle allocation = darea.Allocation;
  // Creamos un pixmap del tamaño del área
  pixmap = new Gdk.Pixmap ( window, allocation.Width,
allocation.Height, -1);

  // Ponemos un cuadro blanco en el pixmap
  pixmap.DrawRectangle (darea.Style.WhiteGC, true, 0, 0,
allocation.Width, allocation.Height);
}

static void Expose_Event (object obj, ExposeEventArgs args)
{
  // Evento que mostrará el pixmap a la drawing area.
  Gdk.Rectangle area = args.Event.Area;
  args.Event.Window.DrawDrawable (darea.Style.WhiteGC, pixmap,
area.X, area.Y, area.X, area.Y, area.Width, area.Height );
}
```

Cairo

Cairo es una biblioteca de imágenes 2D vectoriales que nos permite tener distintos motores para su visualización. Existen motores implementados sobre OpenGL para poder utilizar la aceleración gráfica y disponer de efectos.

Para poder trabajar con Cairo desde GTK#, debemos hacerlo con la clase `Graphics`, que puede ir asociada a una `Surface`. Una `Surface` puede ser una ventana, un *buffer* o un archivo de disco.

Para poder obtener el objeto `Graphics` de un elemento `Drawable` (por ejemplo `DrawingArea`), debemos convertir el área de dibujo a un `Graphics` de Cairo:

```
// Para poder hacer esto se requiere la versión superior a 2.8 de
Gdk.
// Debemos comprobar que tenemos el mapeo de la biblioteca
[DllImport("libgdk-x11-2.0.so")]
internal static extern IntPtr gdk_cairo_create (IntPtr raw);

// Esta función obtendrá el Graphics de Cairo de un Drawable
public static Cairo.Graphics CreateDrawable (Gdk.Drawable
drawable)
{
    Cairo.Graphics g = new Cairo.Graphics (gdk_cairo_create
(drawable.Handle));
    if (g == null)
        throw new Exception ("Couldn't create Cairo Graphics!");

    return g;
}
// Evento de exposed donde vamos a dibujar el contenido
void OnDrawingAreaExposed (object o, ExposeEventArgs args)
{
    DrawingArea area = (DrawingArea) o;
    // Obtenemos el Graphics de Cairo
    Cairo.Graphics g = Graphics.CreateDrawable (area.GdkWindow);

    // Dibujamos

    // Tenemos que liberar la memoria usada por Cairo.
    // En la versión del libro Mono.Cairo no se sincroniza con el
Garbage Collector
   ((IDisposable) gr.Target).Dispose ();
    ((IDisposable) g).Dispose ();
}
```

Una vez que tenemos el `Graphics` de Cairo, podemos dibujar con las funciones de Cairo.

Las primitivas básicas son:

- `PointD (x,y)`: objeto que define un punto en el área.
- `Graphics.MoveTo(PointD)`: función para movernos a un punto determinado.
- `Graphics.LineTo(PointD)`: función para definir una línea desde el punto actual al nuevo.

- `Graphics.CurveTo(PointD, PointD, PointD)`: función para hacer una curva de Bezier desde el punto actual al punto3 usando como puntos de control el punto1 y el punto2.
- `Graphics.ClosePath()`: cierra la línea definida.

Una vez que tenemos las líneas definidas, podemos llenar de un color:

```
Graphics.Color = new Color(0,0,0);
Graphics.FillPreserve();
```

Y dibujar la línea definida:

```
Graphics.Color = new Color(1,0,0);
Graphics.Stroke();
```

Debe tenerse en cuenta que, para usar un color, primero tenemos que definirlo en la propiedad `color` y luego llamar la función que lo usará. El uso de `Fill` o `FillPreserve` y `Stroke` o `StrokePreserve` se diferencia en conservar o no el `Path` creado.

Hay tres propiedades que pueden grabarse temporalmente con la función `Save` y `Restore` para poder modificar temporalmente:

- `Color`: el color,
- `LineWidth`: el tamaño de la línea,
- `LineCap`: el dibujo de la punta de la línea.

A partir de estas funciones básicas y muchas otras que aporta la API de Cairo, como dibujar gradientes o el uso de transparencias, podemos dibujar lo que queramos en nuestra `DrawingArea`.

Pango

Pango es una biblioteca para trabajar con texto en un área `Drawing` de Gdk. Esta biblioteca es mucho más rápida que Cairo, pero también más compleja. Para poder mostrar texto, en ella tenemos que crear un *layout* y decirle al `ExposeEvent` que muestre éste.

```
using System;
using Gtk;

public class MyWindow
{

  Pango.Layout layout;
```

```
  Gtk.DrawingArea da;
  int width = 400;
  int height = 300;

  static void Main()
  {
    Application.Init();
    new MyWindow();
    Application.Run();

  }

  public MyWindow ()
  {

      Window w = new Window("pango");
    w.SetDefaultSize (width, height);
    w.DeleteEvent += new DeleteEventHandler (OnMyWindowDelete);

       // Creamos un área de dibujo
    da = new Gtk.DrawingArea();
    da.SetSizeRequest(width, height);
    da.ExposeEvent += Expose_Event;

       // Creamos un layout de Pango
       layout = new Pango.Layout(w.PangoContext);
       // Definimos el tamaño dado unos píxeles
       layout.Width = Pango.Units.FromPixels(300);
       // Definimos el wrapping en la palabra
       layout.Wrap = Pango.WrapMode.Word;
       // Definimos el alineamiento a la izquerda
       layout.Alignment = Pango.Alignment.Left;
       // Definimos el tipo de fuente
       layout.FontDescription =
Pango.FontDescription.FromString("Ahafoni CLM Bold 100");
      // Definimos el contenido del layout con la palabra "hola"
de color azul
       layout.SetMarkup("<span color=" + (char)34 + "blue" +
(char)34 + ">" + "Hola" + "</span>");

    w.Add(da);
    w.ShowAll ();
  }

  void Expose_Event(object obj, ExposeEventArgs args){
       // Funciona para dibujar el layout en la drawing area
    da.GdkWindow.DrawLayout (da.Style.TextGC (StateType.Normal),
5, 5, layout);
  }

  void OnMyWindowDelete (object sender, DeleteEventArgs a)
  {
    Application.Quit ();
    a.RetVal = true;
  }
}
```

3.3. Uso

Ya hemos visto muchos posibles usos de la biblioteca GTK# en los ejemplos anteriores. Pero hay un par de cosas que deben quedar claras para su uso correcto: el bucle de eventos principal y el *drag and drop* para mover información de un lado para otro de la aplicación.

3.3.1. Bucle de eventos

El modo de funcionar GTK# es a partir de eventos; para eso hay tres funciones que controlan cualquier aplicación usando esta biblioteca:

- **`Application.init`**`()`: esta función permite inicializar toda la biblioteca GTK#. Después de ésta, podemos definir toda la estructura de ventanas o usar la biblioteca Glade para obtenerla.

- **`Application.run`**`()`: cuando ya hemos definido toda la interficie gráfica, podemos ejecutar esta función. En el momento en que se ejecuta, se interrumpe el programa y ya no se ejecuta lo que pueda venir a continuación. A partir de esta función, el programa sólo está atento a los distintos eventos que se ejecutan para ir llamando todas las funciones de *callback*.

- **`Application.quit`**`()`: función que llamamos para terminar el bucle de eventos y salir de la aplicación.

Este sistema de funcionamiento tiene un problema: en el momento de atender un evento, el programa se bloquea y no redibuja los distintos elementos ni puede atender nuevos eventos. En el caso de tener eventos que puedan tardar y bloquear la aplicación, podemos usar *threads* para redibujarla. El problema de usar *threads* es que la biblioteca no es *thread-safe*, con lo que gestionar la interfaz desde un `thread` puede hacer terminar el programa con un error. Para solventar esto tenemos distintas herramientas:

Idle

Se puede definir un trozo de código que se ejecuta cuando el sistema no tiene ningún evento que atender.

```
void Start ()
   {
       // Definimos la función que ejecutaremos cuando no tenga nada que hacer
       GLib.Idle.Add (new IdleHandler (OnIdleCreateThumbnail));
   }
```

Timeouts

Podemos definir un `timeout` para poder actuar si, al pasar un tiempo, el programa no ha reaccionado ante un evento.

```
void StartClock ()
    {
        // Cada 1000 milisegundos ejecutamos la función update_status
```

```
        GLib.Timeout.Add (1000, new GLib.TimeoutHandler (update_status));
    }

    bool update_status ()
    {
        // Cambiamos el texto de un label con la hora.
        time_label.Text = DateTime.Now.ToString ();

        // Si devolvemos false, el timeout dejará de volver a ejecutarse
        // Si devolvemos true, el timeout volverá a cargarse y ejecutará otra vez la función

        return true;
    }
```

Invoke

Podemos ejecutar una función para invocar algún cambio en la interfaz gráfica desde un `thread`.

```
using Gtk;

class TrabajoDuro {
  static Label label;

  static void Main ()
  {
    Application.Init ();
    Window w = new Window ("Cray en una ventana");
    label = new Label ("Computando");

    // Lanzamos un thread que hará los cálculos de gran coste
    Thread thr = new Thread (new ThreadStart (ThreadRoutine));
    thr.Start ();

    Application.Run ();
  }

  static void ThreadRoutine ()
  {
        // Cálculo largo
        LargeComputation ();
      // Invocamos a Gtk para delegarle el cambio del texto del label
        Gtk.Application.Invoke (delegate {
              label.Text = "Done";
        });
  }

  static void LargeComputation ()
  {
    // Cálculos largos
  }
}
```

ThreadNotify

Otra forma de hacer lo mismo es usando `ThreadNotify`. De este modo, creamos una notificación de un `thread` y desde éste lo llamamos cuando finalizamos el trabajo.

```
using Gtk;

class TrabajoDuro2 {

  static ThreadNotify notify;
  static Label label;

  static void Main ()
  {
    Application.Init ();
    Window w = new Window ("Cray en una ventana");
    label = new Label ("Computando");

    // Creamos un thread para hacer un cálculo
    Thread thr = new Thread (new ThreadStart (ThreadRoutine));
    thr.Start ();
    // Creamos un threadnotify con la función ready
    notify = new ThreadNotify (new ReadyEvent (ready));
    Application.Run ();
  }

  // Función que se ejecutará cuando el thread llame al notify.
  static void ready ()
  {
      label.Text = "Done";
  }

  static void ThreadRoutine ()
  {
        // Hacemos los cálculos largos
        LargeComputation ();
        // Levantamos el notify
        notify.WakeupMain ();
  }

  static void LargeComputation ()
  {
    // lots of processing here
  }
}
```

3.3.2. *Drag and drop*

Drag and drop es un método efectivo para permitir al usuario manipular los datos, dentro de la aplicación y entre distintas aplicaciones. En GTK# tenemos un origen de *drag* donde podemos definir los tipos que pueden proveer y los destinos de *drop* que describen qué tipos lo aceptan. Si se producen las dos cosas, se puede producir un *drag and drop*.

Para poder definir que un objeto es destino de un *drag and drop*, vamos a tener que usar la función:

```
...
// Definimos la tabla de tipo de elementos que podemos recibir.
private static Gtk.TargetEntry[] target_table =
       new TargetEntry [] {
          new TargetEntry ("text/uri-list", 0, 0),
          new TargetEntry ("application/x-monkey",0,1),
       };
```

```
....

// Definimos qué podemos recibir en qué widget y cuál será la
acción del drag and drop
Gtk.Drag.DestSet (widget,DestDefaults.All, target_table,
Gdk.DragAction.Copy);
// Creamos un evento para tratar lo que recibimos
widget.DragDataReceived += Data_Received;

....

// Función para recibir el drag
static void Data_Received ( object o , DragDataReceivedArgs args )
{
  bool success = false

  // Obtenemos los datos seleccionados
  string data = System:Text.Encoding.UTF8.GetString (
args.SelectionData.Data );

  // Es una uri-list o un x-monkey
  switch (args.Info) {

    case 0: // uri-list
      // Mostramos el contenido de la uri-list
      string [] uri_list = Regex.Split (data, "\r\n");
      foreach (string u in uri_list) {
        if (u.Length>0)
          System.Console.WrilteLine ("Url : {0}",u);
      }
      success = true;
      break;

    case 1: // x-monkey
      // Mostramos los datos enviados
      System.console.WriteLine("Monkey '{0}'", data);
      success = true;
      break;
  }
  //Finalizamos el drag and drop
  Gtk.Drag.Finish (args.Context,success, false, args.Time);
}
```

Para poder enviar tenemos que usar la siguiente estructura:

```
....
// Definimos el tipo de datos que usaremos al enviar
private static Gtk.TargetEntry [] source_table =
   new TargetEntry [] {
       new TargetEntry ("application/x-monkey", 0, 0),
   };
...

// Definimos los datos que se pueden enviar y el widget de origen
Gtk.Drag.SrouceSet(widget, Gdk.ModifierType.Button1Mask,
source_table, DragAction.Copy);
// Añadimos la función que definirá los datos que se enviarán
widget.DragDataGet += Data_Get;
// Añadimos la función que se ejecutará cuando empecemos el drag
and drop
widget.DragBegin += Drag_Begin;
 ...

static void Data_Get (object o, DragDataGetArgs args)
{
```

```
  // Obtenemos el espació donde vamos a enviar la información
  Atom [] targets = args.Context.Targets;

  // Añadimos la información que queremos enviar
  args.SelectionData.Set (targets [0], 8,
System.Text.Encoding.UTF8.GetBytes("Rupert"));
}

static void Drag_Begin (object o, DragBeginArgs args)
{
  // Cargamos un pixbuf cargado con anterioridad como icono del
cursor
  Gtk.Drag.SetIconPixbuf (args.Context, dibujo, 0, 0);
}
```

4. Glade

La herramienta más extendida para desarrollar la interfaz gráfica de usuario en GTK+ es Glade. No se trata de un entorno de desarrollo completo (IDE), sino de una herramienta con la que gestionar fácilmente el código encargado de crear la GUI. Posteriormente, debemos vincular el resto de código de nuestra aplicación a la capa gráfica generada con Glade.

4.1. Instalación

Actualmente, Glade está disponible en los repositorios de prácticamente la totalidad de las distribuciones, así que su instalación puede hacerse cómodamente mediante el sistema de gestión de paquetes.

Como con el resto de las aplicaciones con licencia GPL, tenemos su código fuente a disposición, y por lo tanto es posible que deseemos compilar las fuentes por nuestra cuenta. En ese caso, deberemos descargarnos las fuentes desde la página web del proyecto y posteriormente las descomprimimos en nuestro espacio de disco. Como es habitual, primero ejecutaremos `./configure` y, al acabar, `make` para compilar el código. Si todo ha funcionado correctamente, podemos pasar a modo superusuario y ejecutar `make install` para finalizar la instalación.

Web recomendada

Podéis acceder a la página web del proyecto en la siguiente dirección <http://glade.gnome.org>.

Una vez que hayamos instalado Glade mediante el sistema de gestión de paquetes o bien a partir del código fuente, podemos proceder a su ejecución con el comando `glade-2`.

4.2. Glade

Existen dos formas de trabajar con Glade:

- usarlo para generar el código fuente en los lenguajes soportados para crear la interfaz, para posteriormente incorporar el código obtenido a nuestro proyecto y rellenar las funciones de gestión de eventos;
- o bien usarlo para crear un archivo de definición de la interfaz en formato XML, para posteriormente cargarlo en tiempo de ejecución desde el código de nuestra aplicación y vincular dinámicamente nuestras funciones de gestión de eventos.

Observación

En este curso no podremos usar la primera opción, ya que a fecha de junio del 2006 Glade no soporta ninguno de los lenguajes disponibles para Mono, en concreto el que estamos utilizando, C#. De este modo, nos centraremos en la segunda alternativa de trabajo.

El hecho de realizar todo el diseño de la interfaz con una herramienta *ad hoc* como Glade, así como poderlo cargar y vincular dinámicamente en tiempo de ejecución, nos brinda mucha flexibilidad. Entre otras cosas, nos será posible

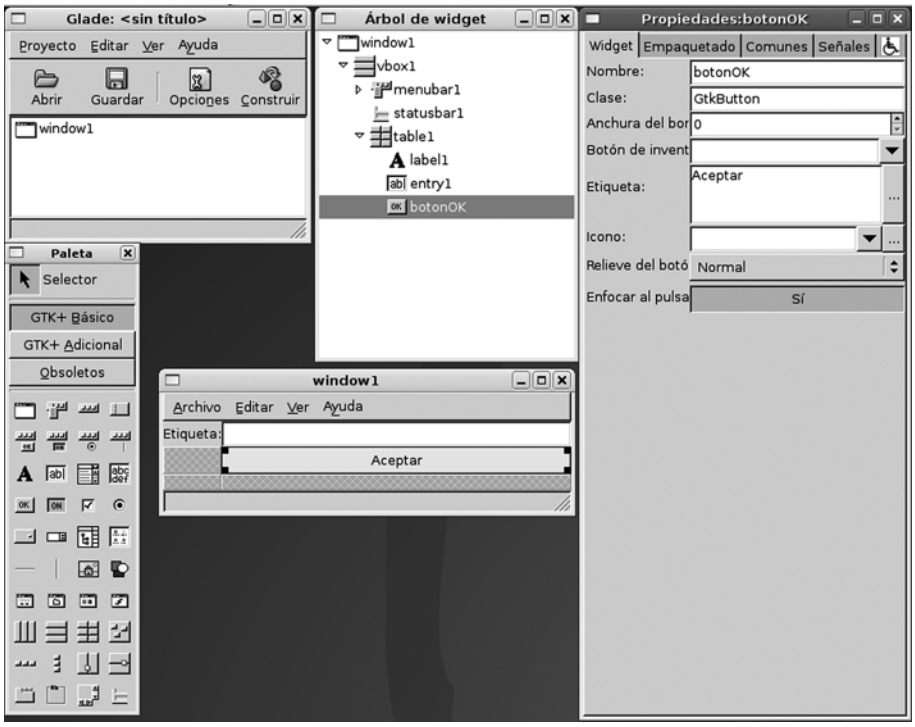
modificar parte de la interfaz gráfica sin necesidad de recompilar el código fuente de nuestra aplicación.

Veamos primero cuál es el modo de diseño de Glade y posteriormente cómo es el archivo XML que nos genera.

La filosofía de trabajo de Glade parte de la base de un sistema de ventanas independientes compuestas por (figura 5):

1) la ventana principal, que nos permite crear, cargar y almacenar el proyecto actual;
2) la paleta, que contiene los elementos gráficos (*widgets*) que podemos insertar en la interfaz;
3) el árbol de elementos, que nos muestra la jerarquía de los elementos que componen la interfaz;
4) las propiedades, que nos permite visualizar y editar las propiedades del elemento gráfico que tengamos seleccionado.

Figura 5



Primero deberemos crear una ventana sobre la cual podremos disponer otros elementos gráficos. Para hacerlo, es suficiente con pulsar el primer botón de la Paleta llamado "Ventana". Automáticamente se creará un elemento gráfico de primer nivel en el Árbol de elementos que será la ventana creada y ésta se mostrará como una ventana más del entorno Glade.

A partir de este punto, iremos escogiendo elementos gráficos de la paleta y los depositaremos en algún otro elemento tipo contenedor del que dispongamos. Por ejemplo, podemos seleccionar el elemento "Caja Vertical" e insertarlo en la ventana. En ese momento, podremos escoger el número de divisiones verticales que

deseamos crear. Supongamos que creamos tres para crear las zonas de menú, contenido y estado típicas de una aplicación convencional. Una vez que hayamos finalizado su creación, veremos las tres divisiones verticales en la ventana de la aplicación. Será necesario escoger los elementos gráfico "Barra de menú" y "Barra de estado", e insertarlos en la primera y última división vertical. Finalmente, en la división central situaremos el resto de los elementos de nuestra aplicación ejemplo. Vamos a crear un contenedor "Tabla" de tres filas y dos columnas. En la primera celda insertaremos una "Etiqueta" y en la segunda una "Entrada de texto". Para acabar, en la celda central derecha colocaremos un "Botón".

Una vez que hemos dispuesto todos los elementos gráficos de nuestro interfaz, es posible modificar sus propiedades mediante la ventana Propiedades. Por ejemplo, podemos cambiar el texto que se muestra al usuario, la propiedad "Etiqueta"; o también podemos cambiar el identificador interno con el que deberemos hacer referencia al elemento gráfico desde nuestro código de programa, la propiedad "Nombre".

Después de crear la interfaz intuitivamente, como hemos visto, lo guardaremos en formato XML mediante la opción "Guardar" de la entrada del menú "Proyecto".

Si analizamos el contenido del archivo XML, podremos observar la estructura siguiente:

```
<?xml version="1.0" standalone="no"?> <!--*- mode: xml -*-->
<!DOCTYPE glade-interface SYSTEM "http://glade.gnome.org/glade-
2.0.dtd">

<glade-interface>

<widget class="{clase}" id="{identificador}">
  <property name="{nombre}">{valor}</property>
  ...
  <child>
    {elementos gráficos contenidos}
 </child>
</widget>

</glade-interface>
```

A partir de aquí, pasaremos al siguiente punto, que será gestionar este archivo desde el código de nuestra aplicación. Aunque, si en algún momento necesitamos retocar su contenido, podremos hacerlo directamente sobre el XML o bien usar de nuevo Glade para hacerlo más cómodamente.

4.3. Glade#

Una vez que hemos definido la estética de la interfaz gráfica de usuario con Glade y por lo tanto disponemos del archivo XML que contiene su codifica-

ción, el siguiente paso será manejarlo desde el código de nuestro programa. Para hacerlo, usaremos la biblioteca Glade#. Su uso más sencillo consiste en la creación de un objeto `Glade.XML` a partir del archivo XML y posteriormente vincular automáticamente todos los elementos gráficos `[Widget]` que hayamos declarado en nuestro código. A partir de este momento, podemos trabajar con los objetos vinculados a los elementos gráficos como si los hubiéramos creado explícitamente en nuestro código.

Veamos un ejemplo sencillo de una aplicación que muestra el área de un rectángulo a partir de los datos de base y altura que el usuario introduce en dos cajas de texto:

```
using System;
using Gtk;
using Glade;
public class EjemploGlade
{
  [Widget] Window ventana;
  [Widget] Button bCalcular;

  [Widget] Entry tBase;
  [Widget] Entry tAltura;
  [Widget] Entry tArea;

  public static void Main (string[] args)
  {
    new EjemploGlade (args);
  }

  public EjemploGlade (string[] args)
  {
    Application.Init();

    Glade.XML gxml = new Glade.XML (null, "guiEjemploGlade.glade",
"ventana", null);
    gxml.Autoconnect (this);

    bCalcular.Clicked   += EventoBotonPulsado;
    ventana.DeleteEvent += new
DeleteEventHandler(EventoCerrarVentana);

    Application.Run();
  }

  public void EventoBotonPulsado(object obj, EventArgs args)
  {
    int resultado  = Int32.Parse(tBase.Text) *
Int32.Parse(tAltura.Text);
    tArea.Text     = resultado.ToString();
  }

  static void EventoCerrarVentana(object obj, DeleteEventArgs args)
  {
    Application.Quit();
  }
}
```

Podemos observar cómo el constructor de nuestra clase aplicación construye el objeto `Glade.XML` a partir del archivo `"guiEjemploGlade.glade"`, usando como ventana inicial la etiquetada con `"ventana"`. Posteriormente, se llama al

método `Autoconnect` de dicho objeto que vincula todos los miembros del objeto aplicación que tienen el atributo `[Widget]`. Por último, se indican qué métodos deben usarse para los eventos de botón pulsado y cerrar aplicación.

Para poder compilar el código C# que usa Glade# es necesario incluir la referencia al espacio de nombres 'glade' que está en el paquete glade-sharp y el archivo XML de definición.

```
mcs -pkg:glade-sharp -resource:guiEjemploGlade.glade EjemploGlade.cs
```

De esta forma, el ejecutable final incorpora el archivo XML en su interior. No obstante, puede sernos interesante vincularlo pero no incluirlo, de forma que tendremos que distribuir el archivo XML junto al ejecutable. Si optamos por esta opción, podremos alterar parte de la interfaz gráfica modificando el archivo XML y sin necesidad de recompilar de nuevo la aplicación.

```
mcs -pkg:glade-sharp -linkresource:guiEjemploGlade.glade EjemploGlade.cs
```

4.4. Manejo de eventos

La programación de la interfaz gráfica siguiendo el paradigma usado por Glade está orientada a eventos. De este modo, el código de control de los elementos gráficos se dedica a reaccionar a los eventos que se producen sobre ellos. Al trabajar con el lenguaje C#, el mecanismo que se usa para acontecer dicha función son los delegados. Éstos nos permiten subscribir más de un método, que será llamado en el momento que se ejecute el delegado.

Sobre los delegados podéis ver el apartado 2.

En el ejemplo del subapartado anterior ya se realizaban estos pasos de forma explícita en el método constructor, pero existe una forma más sencilla de enlazar los métodos de manejo de eventos cuando se trabaja en Glade: se trata de declarar las señales que puede recibir cada elemento gráfico. Para hacerlo, basta con situarse en la pestaña "Señales" de la ventana de Propiedades del Glade. Desde ella podemos escoger el nombre de la función que debe suscribirse a cada uno de los eventos posibles. Esta información es guardada en el archivo XML:

```
<signal name="{evento}" handler="{método}" />
```

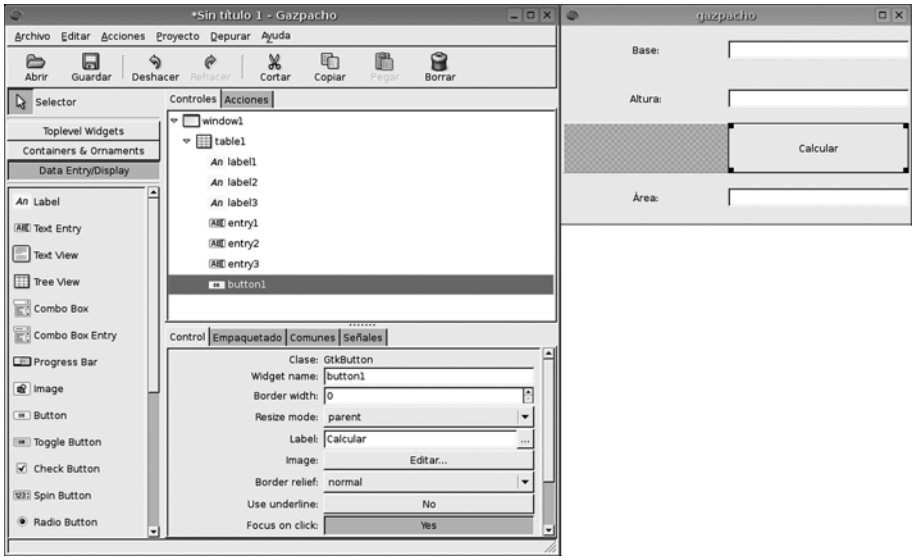
Posteriormente, al cargar el archivo XML de Glade desde el programa y usar el método `Autoconnect`, además de vincular todos los *widgets* declarados en el código con los elementos gráficos, también subscribirá todos los métodos con los nombres indicados a los eventos correspondientes.

4.5. Alternativas: Gazpacho y Stetic

Existen varias alternativas a Glade para diseñar la interfaz gráfica de una aplicación, entre ellas Gazpacho y Stetic.

Gazpacho se trata de una aplicación desarrollada en Python que genera archivos en el formato XML de Glade, pero con una interfaz ligeramente diferente. En lugar de mantener por separado todas las ventanas del diseñador, éstas están todas integradas en una sola como se muestra en la figura 6.

Figura 6

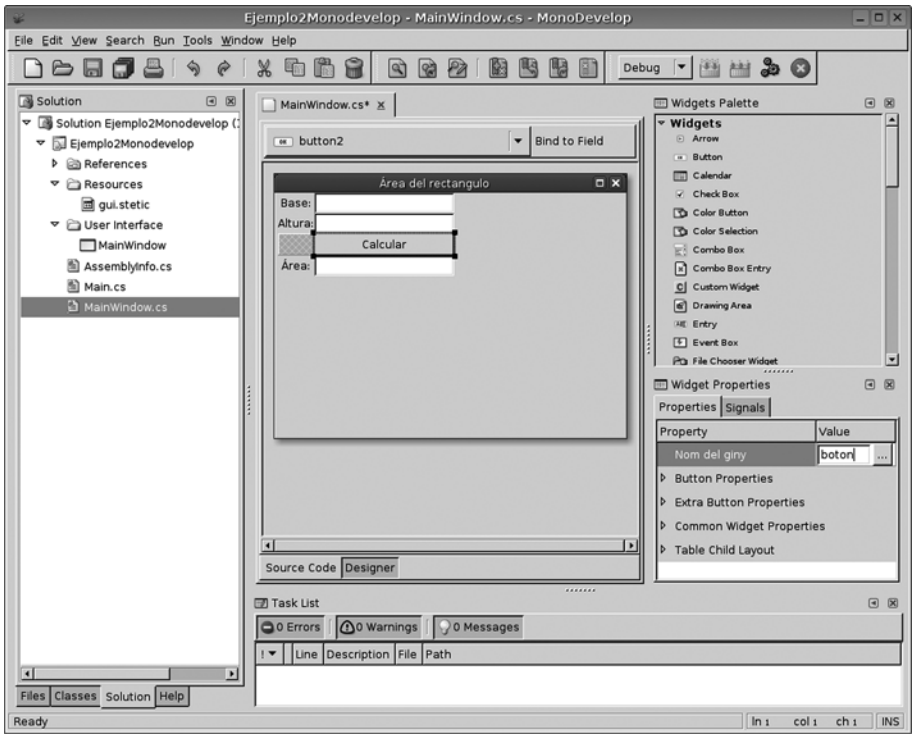


Sus creadores se muestran convencidos de que esta distribución es mucho más cómoda. Además, incorpora otras posibilidades con el ánimo de mejorar la herramienta, como por ejemplo el uso de plantillas para elementos gráficos. Esta herramienta permite definir un nuevo elemento disponible para ser insertado en la interfaz a partir de cualquiera que ya tengamos diseñada, de modo que podemos crearnos nuestra propia biblioteca de elementos para la interfaz. Algunas de estas características, como las plantillas, se están incorporando en futuras versiones de Glade.

La aplicación **Stetic** es muy similar a Gazpacho. La diferencia principal reside en que Stetic ha sido empotrado en Monodevelop, hecho que facilita el diseño simultáneo de la interfaz y el código interno de nuestra aplicación. Al usar el diseñador Stetic empotrado, Monodevelop genera automáticamente dos archivos: gui.stetic y generated.cs. El primero es un archivo de formato compatible con Glade, en el que se almacena el diseño creado. El segundo archivo se genera a partir del primero y corresponde al código necesario para crear la interfaz en C#. De este modo, el funcionamiento corresponde a la alternativa del expuesto en los subapartados anteriores. Es decir, en lugar de generar la interfaz en tiempo de ejecución, a partir de su archivo de definición XML, es generado en tiempo de compilación. Por

otra parte, la integración del diseñador de interfaz con el resto del IDE permite otras comodidades, como el hecho de que, al crear "señales", se insertan directamente los métodos correspondientes en el código para que podamos rellenarlos.

Figura 7



Web recomendada

Tenéis más información en <http://pixane.net/blog/?p=47>.

5. *Widgets* avanzados

5.1. GtkHtml#

El primer elemento gráfico avanzado que veremos es GtkHtml#, que es un elemento para visualizar y editar HTML en nuestras aplicaciones. Se trata de un elemento sencillo y eficaz con soporte básico de HTML que no incluye otras tecnologías como hojas de estilo (CSS) ni JavaScript. Si necesitamos más prestaciones, tendremos que recurrir al Gecko#, que se expone en el subapartado siguiente.

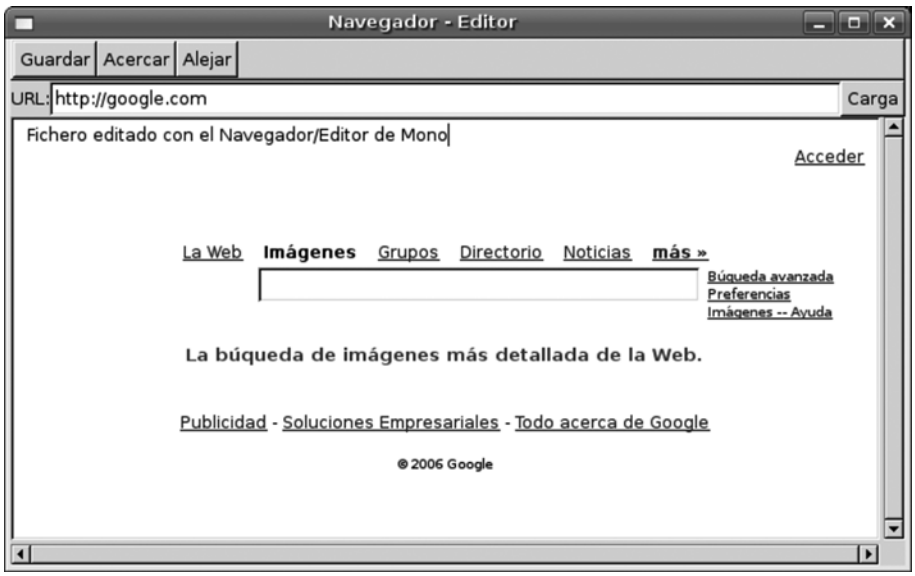
Entre los distintos miembros de la clase GtkHtml#, destacamos los siguientes para proporcionar una idea de sus prestaciones:

- Propiedades:
  - `Editable`: commutar a cierto para permitir al usuario modificar el contenido.
  - `InlineSpelling`: commutar a cierto para activar la corrección ortográfica.
- Métodos:
  - `Begin()`, `Write(HTMLStream, string, int)`, `End(HTMLStream, HTMLStreamStatus)`: cárgar contenido HTML a partir de un *stream* de entrada.
  - `Save(HTMLSaveReceiverFn)`, `Export(string, HTMLSaveReceiverFn)`: extraer el contenido HTML del *widget*.
  - `Print(Gnome.PrintContext)`: imprimir el contenido HTML.
  - `ZoomIn()`, `ZoomOut()`, `ZoomReset()`: modificar la escala de visualización.
  - `Copy()`, `Cut()`, `Paste(bool)`, `Undo()`, `Redo()`: operaciones básicas de edición.
  - `SelectAll()`, `SelectLine()`, `SelectParagraph()`, `SelectWord()`, `GetObjectById(string)`: métodos para seleccionar fragmentos de HTML.
  - `InsertHtml(string)`, `AppendHtml(string)`: para añadir código HTML en la posición actual del cursor o al final de todo el contenido.
  - `Command(string)`: ejecución de comandos como por ejemplo
    - `Search`: buscar una cadena dentro del contenido HTML.
    - `InsertParagraph`: insertar un párrafo.
    - `MakeLink`: insertar un enlace.
    - `BoldOn`, `ItalicOn`, `SizeIncrease`, `AlignCenter`, `ParagraphStyleNormal`, `ParagraphStyleH1`, etc.: modificar el estilo del texto.
    - `InsertTable11`, `TableInsertRowBefore`, `TableJoinCellLeft`, etc.: crear y manipular tablas.
    - `CursorBod`, `CursorEod`: situar el cursor al principio / final del documento.
- Eventos:
  - `CursorMove`: al mover el cursor.

- `LinkClicked`: al pulsar sobre un enlace.
- `LoadDone`: al finalizar la carga del documento.
- `Submit`: al enviarse los datos de un formulario.

A continuación, tenemos una aplicación de ejemplo para ilustrar el funcionamiento básico de GtkHtml#. La aplicación es un navegador de páginas en HTML de Internet que permite modificarlas y guardarlas en disco, tal como muestra la figura 8.

Figura 8



La interfaz consiste en una caja de texto en la que entrar la URL que deba cargarse, unos botones para guardar el documento en un archivo local y modificar la escala de visualización, y finalmente el área GtkHtml# en la que se visualiza el documento y se permite modificarlo con las teclas habituales de edición. Esta interfaz ha sido creada mediante Glade y se carga a partir del archivo XML que este programa de diseño genera.

El siguiente listado corresponde al código de la aplicación:

```
using System;
using System.Net;
using System.IO;
using Gtk;
using Glade;

namespace EjemploGtkHtml
{
  class EjemploGtkHtml
  {
    [Widget] Window windowPrincipal;
    [Widget] Button buttonGuardar;
    [Widget] Button buttonAcercar;
    [Widget] Button buttonAlejar;
    [Widget] Button buttonCarga;
    [Widget] Entry  entryURL;
    [Widget] ScrolledWindow scrolledwindowTrabajo;
```

```
HTML         html;
string       actualURL;
StreamWriter archivoSalida;

static void Main (string[] args) { new EjemploGtkHtml(); }

EjemploGtkHtml()
{
  Application.Init ();

  Glade.XML gxml = new Glade.XML (null, "ejemplogtkhtml.glade",
                                  "windowPrincipal", null);
  gxml.Autoconnect (this);

  windowPrincipal.DeleteEvent += new DeleteEventHandler(on_windowPrincipal_delete);

  html = new HTML ();
  html.LinkClicked += new LinkClickedHandler (on_html_linkclicked);
  scrolledwindowTrabajo.Add (html);

  LoadHtml("http://tornatmico.org");
  windowPrincipal.ShowAll();
  Application.Run ();
}

void on_windowPrincipal_delete (object obj, DeleteEventArgs args)
{ Application.Quit(); }

void on_entryURL_activate (object obj, EventArgs args)
{ on_buttonCarga_clicked (obj, args); }

void on_buttonAcercar_clicked (object obj, EventArgs args) { html.ZoomIn(); }

void on_buttonAlejar_clicked (object obj, EventArgs args) { html.ZoomOut(); }


void on_buttonCarga_clicked (object obj, EventArgs args)
{
  actualURL = entryURL.Text.Trim();
  LoadHtml (actualURL);
}

void on_html_linkclicked (object obj, LinkClickedArgs args)
{
  string nuevaURL;

  if (args.Url.StartsWith("http://"))
    nuevaURL = args.Url;
  else
    nuevaURL = actualURL + args.Url;

  try { LoadHtml (nuevaURL); } catch { }
  actualURL = nuevaURL;
}

void on_buttonGuardar_clicked (object obj, EventArgs args)
{
  archivoSalida = File.CreateText("archivoSalida.html");
  html.Export("text/html", new HTMLSaveReceiverFn( GuardaEnDisco ) );
  archivoSalida.Close();
}

bool GuardaEnDisco (IntPtr obj, string data)
{
  archivoSalida.Write(data);
  return true;
}
```

```
    void LoadHtml (string URL)
    {
      HttpWebRequest web_request = (HttpWebRequest) WebRequest.Create (URL);
      HttpWebResponse web_response = (HttpWebResponse) web_request.GetResponse ();
      Stream stream = web_response.GetResponseStream ();
      byte [] buffer = new byte [8192];

      html.Editable = false;
      HTMLStream html_stream = html.Begin ();
      int count;

      while ((count = stream.Read (buffer, 0, 8192)) != 0){
           html_stream.Write (buffer, count);
      }
      html.End (html_stream, HTMLStreamStatus.Ok);

      html.Editable = true;
      html.InsertHtml("Archivo editado con el Navegador/Editor de Mono");
    }
  }
}
```

En primer lugar podemos observar cómo en el constructor de la clase se carga dinámicamente la definición de la interfaz y posteriormente se le añade el objeto GtkHtml#. De este modo, se muestra cómo puede aprovecharse Glade para todo lo que proporcione directamente, y completar el resultado con la construcción mediante código de los elementos de interfaz todavía no soportados por el diseñador.

El mismo constructor carga una URL inicial usando el método `LoadHtml`, el cual obtiene la información por medio de un agente y la deposita en el *widget* GtkHtml mediante los métodos `Begin()`, `Write()` y `End()`. Al acabar, activa la posibilidad de que el usuario pueda editar el contenido e inserta un fragmento de HTML inicial.

A partir de aquí, el usuario puede manipular el contenido del área GtkHtml con las teclas habituales de edición (cursores, inserción, supresión, ratón, etc.) y modificar la escala de visualización con los eventos `on_buttonAcercar_clicked` / `on_buttonAlejar_clicked`.

Por último, el usuario puede guardar en disco el contenido modificado mediante el botón Guardar, que activa el evento `on_buttonGuardar_clicked`.

Esta aplicación sencilla muestra la base para crear un navegador y/o editor ligero de HTML integrado en nuestra aplicación. Para compilarla será necesario indicar que estamos usuando los paquetes GTK# y GtkHtml#, tal y como se indica a continuación:

```
mcs -pkg:gtk-sharp -pkg:gtkhtml-sharp -pkg:glade-sharp -
resource:ejemplogtkhtml.glade  EjemploGtkHtml.cs
```

Instalación de GtkHtml

Dado que GtkHtml# es sólo una capa de interconexión con GtkHtml, será necesario tener instalado este último. Normalmente, lo encontraremos en el paquete libgtkhtml en una versión superior o igual a la 3.0.

5.2. Gecko#

El proyecto Mozilla ha generado –entre otros– el motor de visualización de páginas web Gecko. Este motor es capaz de visualizar páginas web basadas en HTML o XML, incluyendo CSS, imágenes, JavaScript y componentes que requieran aplicaciones externas, como por ejemplo contenido Flash. Para poder utilizarlo desde el proyecto Mono, existe una capa de interconnexión que permite incorporarlo a nuestra aplicación como un elemento gráfico más: se trata de Gecko#. Este elemento gráfico interactúa con GtkEmbedMoz, subministrado con las aplicaciones principales del proyecto Mozilla.

Gecko# proporciona principalmente el elemento gráfico `WebControl`, del cual podemos destacar los siguientes elementos:

- Propiedades:

– `Location`: dirección actual.
– `Title`: título de la página actual.

- Métodos:

– `GoBack()`: acceder a la página visitada anteriormente.
– `GoForward()`: acceder a la página desde la cual se había retrocedido.
– `LoadUrl(string)`: cargar una nueva página desde Internet.
– `Reload(int)`: recargar la página actual.
– `StopLoad()`: detener la carga de la página actual.

- Eventos:

– `DomKeyPress`: al pulsar una tecla en cualquier parte del documento.

– `DomMouseClick`, `DomMouseDblClick`: al pulsar el ratón en cualquier parte del documento.

– `LinkMsg`: al situar el ratón o el foco del teclado en un enlace (sin necesidad de activarlo).

– `LocChange`: al canviar la dirección actual.

– `NetStart`, `NetStop`: al empezar o terminar la carga del documento actual.

– `TitleChange`: al cambiar el título de la página actual.

A continuación, se expone una aplicación que muestra el funcionamiento básico de Gecko#. Se trata de un navegador básico con las opciones de desplazarse por la historia de navegación, así como recargar o detener la carga de la

página actual. La figura 9 muestra su apariencia, que está definida básicamente mediante un archivo XML usando Glade.

Figura 9

La interfaz consta de una división vertical para situar en la parte superior los botones de navegación, seguidos de la barra de dirección, y en la parte inferior una barra de estado y de progreso. La parte central de la ventana está ocupada por el elemento gráfico `WebControl` de Gecko#, es decir la zona de visualización de la página web.

El listado de la aplicación es el siguiente:

```
using System;
using Gtk;
using Gecko;
using Glade;

namespace EjemploGecko
{
  class EjemploGecko
  {
    [Widget] Window windowPrincipal;
    [Widget] Button bAtras;
    [Widget] Button bAdelante;
    [Widget] Button bParar;
    [Widget] Button bRecargar;
    [Widget] Button bCargar;
    [Widget] Entry  actualURL;
    [Widget] HBox   hboxBottom;
    [Widget] VBox   vboxWebControl;

    WebControl  webControl;
    Statusbar   barraEstado;
```

```
    ProgressBar barraProgreso;

    static void Main (string[] args)
    {
       new EjemploGecko ();
    }

    EjemploGecko ()
    {
      Application.Init ();

      Glade.XML gxml = new Glade.XML (null, "EjemploGecko.glade",
                                      "windowPrincipal", null);
      gxml.Autoconnect (this);

      webControl = new WebControl ("/tmp/csharp", "EjemploGecko");
      webControl.OpenUri += new OpenUriHandler(moz_openuri);
      webControl.LinkMsg += new EventHandler(moz_linkmsg);
      vboxWebControl.PackStart(webControl, true, true, 1);
      webControl.Visible = true;

      barraEstado   = new Statusbar ();
      barraProgreso = new ProgressBar ();

      hboxBottom.Add (barraEstado);
      hboxBottom.Add (barraProgreso);

      windowPrincipal.ShowAll ();
      Application.Run ();
    }

    void LoadHtml      (string URL)                  { webControl.LoadUrl (URL); }

    void bCargar_clk   (object obj, EventArgs args)
       { LoadHtml(actualURL.Text.Trim()); }

    void actualURL_act (object obj, EventArgs args)  { bCargar_clk (obj, args); }

    void bAtras_clk    (object obj, EventArgs args)  { webControl.GoBack(); }

    void bParar_clk    (object obj, EventArgs args)  { webControl.StopLoad(); }

    void bAdelante_clk (object obj, EventArgs args)  { webControl.GoForward(); }

    void bRecargar_clk (object obj, EventArgs args)  { webControl.Reload(0); }

    void moz_openuri   (object obj, OpenUriArgs args){ actualURL.Text = args.AURI; }

    void moz_linkmsg   (object obj, EventArgs args)
      { barraEstado.Pop (1);
        barraEstado.Push (1, webControl.LinkMessage);
      }

    void windowPrincipal_delete(object obj, DeleteEventArgs args)
      { Application.Quit(); }
  }
}
```

Como puede observarse, el método constructor genera la estructura principal de la interfaz cargando un archivo XML de descripción con el formato Glade. A continuación, le añade algunos elementos gráficos de más, como el mismo `WebControl`. Por último, se muestran todos los métodos que responderán a los eventos de la interfaz, los cuales básicamente realizan llamadas a métodos del componente `WebControl`.

Para compilar aplicaciones usando Gecko#, necesitaremos incluirlo como paquete, tal y como muestra el siguiente ejemplo:

```
mcs -pkg:gecko-sharp -pkg:glade-sharp -
resource:EjemploGecko.glade EjemploGecko.cs
```

Instalación de GtkEmedMoz

Dado que Gecko# es sólo una capa de interconexión con GtkEmedMoz, será necesario tener instalado este último. Normalmente lo encontraremos en el paquete Mozilla de nuestra distribución.

5.3. GtkSourceView#

Por último, veremos un elemento gráfico útil para crear editores de texto, se trata de GtkSourceView#. De nuevo se trata de una capa de interconexión para poder incluir desde Mono el elemento `SourceView` de las bibliotecas GTK. Este elemento ofrece las funcionalidades básicas que pueden requerirse en un área de edición de código de programación, y en realidad se compone de varias clases, entre las cuales destacamos:

a) `SourceBuffer`:

- Propiedades:
  - `Text`: el texto mostrado.
  - `CheckBrackets`: si es cierto, se resaltan los paréntesis inicial y final al pasar sobre uno de ellos.
  - `Highlight`: si es cierto, se resalta la sintaxis del código mostrado.
  - `Language`: indica el lenguaje del código editado.
- Métodos:
  - `ForwardSearch`, `BackwardSearch`: para realizar búsquedas en el texto.
  - `CreateMaker`, `GetMaker`, `GetNextMaker`: para seleccionar fragmentos del texto.
  - `Undo`, `Redo`: para deshacer o rehacer las últimas acciones.
- Eventos:
  - `HighlightUpdated`: al modificarse algún parámetro del resaltado de código.
  - `MarkerUpdated`: al modificarse la selección del texto.

b) `SourceView`:

- Propiedades:
  - `AutoIndent`: para activar la indentación automática del código.
  - `InsertSpacesInsteadOfTabs`: usar espacios cuando se pulse la tecla del tabulador.
  - `Margin`: márgenes desde los límites del texto al contorno del área de edición.
  - `ShowLineNumbers`: mostrar los números de línea.
  - `TabsWidth`: anchura de los tabuladores.
- Eventos:
  - `Redo`, `Undo`: al activarse los métodos para deshacer o rehacer las últimas acciones.

Para ejemplificar el uso de GtkSourceView# se ha implementado la mínima aplicación que ofrece las funcionalidades para editar un archivo de código C#. En la figura 10 se muestra la interfaz.

Figura 10

La aplicación se inicia con el área de edición en blanco, con lo cual es posible crear un archivo nuevo y posteriormente pulsar sobre el botón "Guardar" para almacenarlo en disco. También existe el botón "Abrir" por si se desea editar un archivo ya existente. El área de edición permite la interacción mediante el teclado y el ratón, con las operaciones básicas de edición (supresión, selección, portapapeles, etc.).

El código de la aplicación es el siguiente:

```
using System;
using System.IO;
using Gtk;
using Glade;
using GtkSourceView;

class Editor
{
  [Widget] Gtk.Window     windowPrincipal;
  [Widget] ScrolledWindow scrolledwindow;
  [Widget] Button         bAbrir;
  [Widget] Button         bGuardar;
  SourceBuffer            sourceBuffer;
  SourceView              sourceView;

  string archivo = "archivo_ejemplo.cs";

  public Editor()
  {
    Application.Init();

   Glade.XML gxml = new Glade.XML(null, "EjemploGtkSourceView.glade", "windowPrincipal", null );
    gxml.Autoconnect( this );

    windowPrincipal.Resize(300, 500 );
```

```
    SourceLanguagesManager sl = new SourceLanguagesManager();
    sourceBuffer = new SourceBuffer(sl.GetLanguageFromMimeType("text/x-csharp"));
    sourceBuffer.Highlight = true;

    sourceView = new SourceView(sourceBuffer);
    sourceView.ShowLineNumbers = true;
    scrolledwindow.Add( sourceView );

    windowPrincipal.DeleteEvent += new DeleteEventHandler( Salir );
    bAbrir.Clicked              += new EventHandler( Abrir );
    bGuardar.Clicked            += new EventHandler( Guardar );


    windowPrincipal.ShowAll();
    Application.Run();
  }

  void Salir(object o, DeleteEventArgs args) { Application.Quit(); }

  void Abrir (object o, EventArgs args) {
    FileSelection fs = new FileSelection ("Escoge archivo");
    fs.Run ();
    archivo = fs.Filename;
    fs.Hide ();

    using (StreamReader sr = new StreamReader(archivo))
    {
        sourceBuffer.Text = sr.ReadToEnd();
    }
  }

  void Guardar( object o, EventArgs args )
  {
    using (StreamWriter sw = new StreamWriter(archivo))
    {
        sw.Write(sourceBuffer.Text);
    }
  }

  public static void Main() { new Editor(); }
}
```

Al igual que en los ejemplos anteriores, la interfaz se construye a partir de la descripción del archivo XML de Glade y se complementa con elementos gráficos creados en el constructor de la aplicación. Dada la simplicidad del ejemplo, se suscriben principalmente los métodos correspondientes a los dos botones. En éstos se usan `StreamReaders` y `StreamWriters` para trabajar con el sistema de archivos.

Para compilar una aplicación que incluya este elemento gráfico será necesario indicar el paquete. Éste está disponible en la versión 2, así que deberemos incluir todo el entorno en esta versión, tal y como se muestra en el ejemplo:

```
mcs -pkg:gtk-sharp-2.0 -pkg:glade-sharp-2.0 -pkg:gtksourceview-
sharp-2.0 -linkresource:EjemploGtkSourceView.glade
EjemploGtkSourceView.cs
```

6. Gnome#

En apartados anteriores se ha mostrado el modo de programar la interfaz gráfica de una aplicación usando GTK y la herramienta de diseño Glade. Se han visto varios elementos gráficos que se tienen a disposición en GTK y cómo usarlos. De todas maneras, para desarrollar aplicaciones podemos aprovechar el proyecto Gnome. Este proyecto tiene como objetivos principales proporcionar un escritorio intuitivo y fácil de usar, así como un buen entorno de desarrollo. En este sentido, además de los elementos GTK ya vistos, podemos usar los elementos básicos que proporciona Gnome para desarrollar aplicaciones.

En este apartado se exponen varios elementos que proporciona el proyecto Gnome con los que desarrollar más rápidamente nuestras aplicaciones:

- **Elementos gráficos**: elementos para insertar en las ventanas de aplicación. Se trata de elementos complejos creados a partir de elementos gráficos básicos de GTK.

  Ejemplo de elemento gráfico

  Gnome proporciona, por ejemplo, un elemento gráfico para escoger una fuente de escritura, junto con su estilo y tamaño, todo integrado dentro de un solo elemento gráfico.

- **Diálogos**: ventanas de diálogo estándar o personalizadas creadas a partir de elementos gráficos GTK o Gnome para facilitar la interacción con el usuario.

- **Impresión**: gestión del sistema de impresión, con el manejo de dispositivos y la existencia de los diálogos básicos de impresión.

- ***Canvas***: elemento gráfico para insertar en las ventanas de aplicación en el que la aplicación puede dibujar fácilmente elementos geométricos básicos.

- **Asistentes**: Gnome proporciona un método ágil para crear agrupaciones de diálogos con la finalidad de llevar a cabo tareas complejas.

6.1. Elementos gráficos

Gnome proporciona elementos gráficos para insertar en las ventanas de aplicación creados a partir de elementos gráficos básicos de GTK. De este modo, en muchas ocasiones podemos usarlos en lugar de tener que diseñarlos de nuevo a partir de GTK. Además, el uso de los elementos proporcionados por Gnome dota de ma-

yor coherencia entre la interfaz de nuestra aplicación y el resto de aplicaciones Gnome. Este último punto aumenta la usabilidad de nuestro diseño.

Para ilustrar el uso de estos elementos se muestra a continuación un ejemplo que usa varios de ellos:

- **`FileEntry`**: para seleccionar archivos. Permite examinar el sistema de archivos mediante un diálogo estándar de Gnome y al mismo tiempo ofrece una lista desplegable con las últimas selecciones efectuadas.

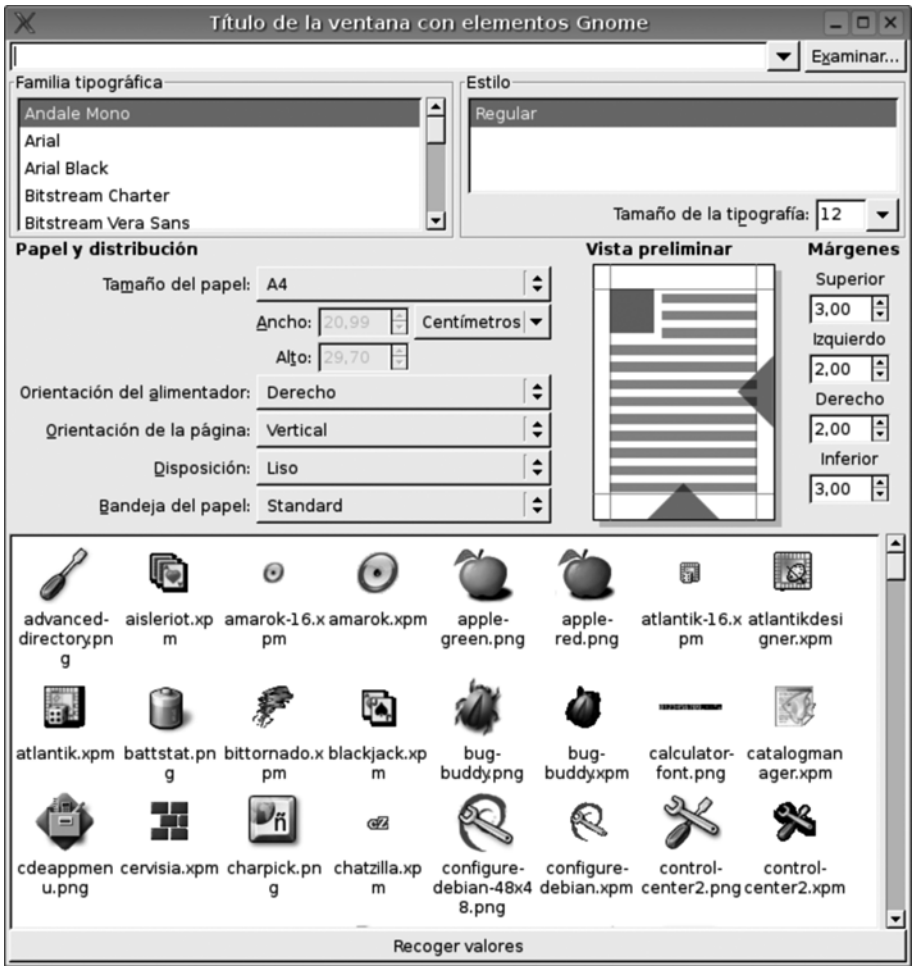
- **`PaperSelector`**: para seleccionar el papel de impresión junto con la disposición del contenido. Este elemento interactúa directamente con el gestor de impresión de Gnome.

- **`FontSelection`**: para seleccionar fuentes de letra, tanto la familia como el estilo y el tamaño. Este elemento interactúa directamente con el gestor de fuentes de Gnome.

- **`IconSelection`**: para seleccionar iconos. Este elemento interactúa directamente con el gestor de iconos de Gnome.

La figura 11 muestra la interfaz de la aplicación ejemplo.

Figura 11



En ella pueden verse todos los elementos gráficos definidos. En general, estos elementos también se encuentran a disposición en las herramientas de diseño de interfaz como Glade, aunque en el código fuente de esta aplicación se muestra cómo crearlos desde el propio código C#:

```
using System;
using Gtk;
using GtkSharp;
using Gnome;

class EjemploGnomeWidgets
{
  Program program;
  Gnome.FileEntry     selectorArchivo;
  Gnome.FontSelection selectorFuente;
  Gnome.PaperSelector selectorPapel;
  Gnome.IconSelection selectorIconos;
  Button              buttonRecoger;

  static void Main(string[] args) { new EjemploGnomeWidgets(args);   }

  EjemploGnomeWidgets (string[] args)
  {
    program = new Program("EjemploGnome", "0.1", Gnome.Modules.UI , args);

    App app = new App("Elementos", "Título de la ventana con elementos Gnome");
    app.DeleteEvent += new DeleteEventHandler (on_app_delete);

    VBox vb = new VBox();

    selectorArchivo = new Gnome.FileEntry ("historial", "título");
    selectorFuente  = new Gnome.FontSelection();
    selectorPapel   = new Gnome.PaperSelector( PrintConfig.Default() );
    selectorIconos  = new Gnome.IconSelection();
    buttonRecoger   = new Button ("Recoger valores");

    selectorIconos.AddDefaults();
    selectorIconos.ShowIcons();

    buttonRecoger.Clicked += new EventHandler (on_buttonRecoger_clicked);

    vb.PackStart( selectorArchivo );
    vb.PackStart( selectorFuente  );
    vb.PackStart( selectorPapel   );
    vb.PackStart( selectorIconos  );
    vb.PackStart( buttonRecoger   );

    app.Contents = vb;
    app.ShowAll();

    program.Run();
  }
        void on_buttonRecoger_clicked(object o, EventArgs args)
  {
    Gnome.Font  fuente   = selectorFuente.Font;

    Console.WriteLine("Archivo: " + selectorArchivo.GetFullPath(true) );
    Console.WriteLine("Fuente: "  + fuente.FamilyName + " " + fuente.Size);
    // see http://cvs.gnome.org/viewcvs/libgnomeprint/libgnomeprint/gnome-print-
config.h?rev=1.29&view=markup
    Console.WriteLine("Papel: "
         + PrintConfig.Default().Get("Settings.Output.Media.PhysicalSize") +
     "(" + PrintConfig.Default().Get("Settings.Output.Media.PhysicalOrientation") + ")");
    Console.WriteLine("Icono: "   + selectorIconos.GetIcon(true) );
  }
  private void on_app_delete (object o, DeleteEventArgs args) { program.Quit (); }
}
```

La creación de los elementos gráficos se realiza en el método principal y consiste en crear los objetos pasando determinados parámetros de personalización a sus métodos constructores. Destacamos el constructor caso del `PaperSelector`, al cual es necesario indicarle cuál de las configuraciones se desea usar, que en el ejemplo es la que tenga el sistema definida por defecto. Por otro lado, cada elemento gráfico tiene sus campos y métodos propios donde se almacena la información recogida. El método `on_buttonRecoger_clicked` ejemplifica cómo recoger dicha información.

6.2. Diálogos

Los diálogos son ventanas o pequeños conjuntos de ventanas pensados para dialogar con el usuario y obtener unos datos concretos. Gnome proporciona diversos diálogos estándar que habitualmente se usan en las aplicaciones. Su uso incrementa la usabilidad de la aplicación ya que el usuario los encuentra familiares.

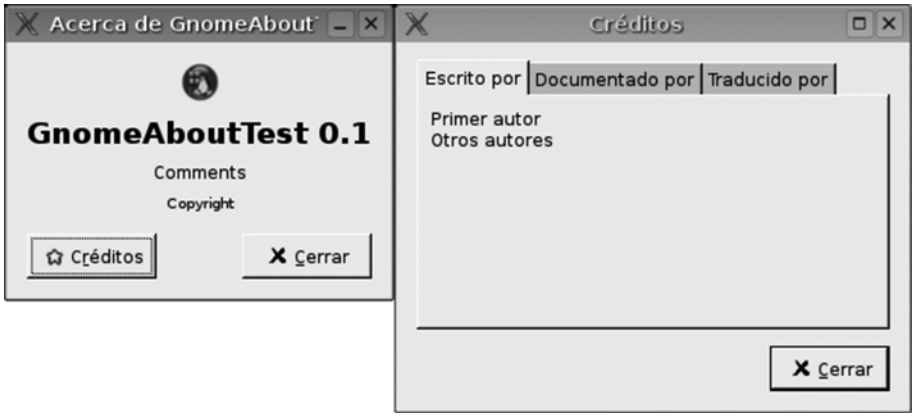
Para ejemplificar su uso, se ha creado una pequeña aplicación que muestra una ventana principal (figura 12) que ofrece diferentes botones para que el usuario pueda llamar a cada uno de los diálogos mostrados.

Figura 12

Los diálogos son éstos:

- **Acerca de** (figura 13): diálogo que muestra los créditos de autoría de la aplicación (figura 13).

Figura 13



- **Autentificación** (figura 14): diálogo para pedir el identificador y la contraseña al usuario. Permite establecer valores para los dos campos, así como identificación anónima.

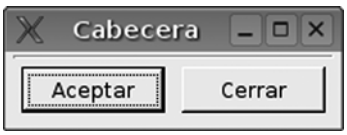
Figura 14

- **Impresión** (figura 15): diálogo para que el usuario pueda especificar determinados parámetros de impresión.

Figura 15

- **Diálogo personalizado** (figura 16): aparte de los diálogos estándar, también es posible crear nuestro propio diálogo personalizado con los elementos que nos interesen.

Figura 16



El código necesario para crear los diálogos que se han mostrado anteriormente es el siguiente:

```
using System;
using Gtk;
using GtkSharp;
using Gnome;

class EjemploGnomeDialogos
{
  Program program;

  static void Main(string[] args) { new EjemploGnomeDialogos(args);   }

  EjemploGnomeDialogos (string[] args)
```

```
{
  program = new Program("EjemploGnome", "0.1", Gnome.Modules.UI , args);

  App app = new App("EjemploGnome", "Ejemplo Gnome");
  app.DeleteEvent += new DeleteEventHandler (on_app_delete);

  HBox hb = new HBox();

  Button buttonAcercaDe       = new Button ( Gtk.Stock.DialogQuestion );
  Button buttonPassword       = new Button ( Gtk.Stock.Network );
  Button buttonPrintConfig    = new Button ( Gtk.Stock.Print );
  Button buttonDialogoPropio  = new Button ( Gtk.Stock.Properties );
  buttonAcercaDe.Clicked      += new EventHandler (on_buttonAcercaDe_clicked);
  buttonPassword.Clicked      += new EventHandler (on_buttonPassword_clicked);
  buttonPrintConfig.Clicked   += new EventHandler (on_buttonPrintConfig_clicked);
  buttonDialogoPropio.Clicked += new EventHandler (on_buttonDialogoPropio_clicked);

  hb.PackStart(buttonAcercaDe);
  hb.PackStart(buttonPassword);
  hb.PackStart(buttonPrintConfig);
  hb.PackStart(buttonDialogoPropio);

  app.Contents = hb;

  app.ShowAll();
  program.Run();
}

private void on_buttonAcercaDe_clicked (object obj, EventArgs args)
{
  string[] authors = {"Primer autor", "Otros autores"};
  string[] documenters = {"Encargado de documentación"};
  Gdk.Pixbuf pixbuf = new Gdk.Pixbuf ("EjemploIcono.png");

  About dialogoAcercaDe    = new Gnome.About ("GnomeAboutTest", "0.1",
                       "Copyright", "Comments",
                       authors, documenters, "translator", pixbuf);
  dialogoAcercaDe.Response += new ResponseHandler (OnResponse);
  dialogoAcercaDe.Run ();
}

private void on_buttonPassword_clicked (object obj, EventArgs args)
{
  PasswordDialog dialogoPwd = new Gnome.PasswordDialog("Título", "Mensage",
                                  "usuario inicial",
                                  "clave inicial", false);
  dialogoPwd.Response      += new ResponseHandler (OnResponse);
  dialogoPwd.Run ();
  Console.WriteLine(dialogoPwd.Username +":"+ dialogoPwd.Password);
  dialogoPwd.Destroy ();
}

private void on_buttonPrintConfig_clicked (object obj, EventArgs args)
{
  PrintConfig conf = PrintConfig.Default();
  PrintConfigDialog dialogoPrtCnf = new Gnome.PrintConfigDialog( conf );
  dialogoPrtCnf.Response         += new ResponseHandler (OnResponse);
  dialogoPrtCnf.Run ();
  Console.WriteLine("Duplex: " + conf.Get("Settings.Output.Job.Duplex"));
  dialogoPrtCnf.Destroy ();
}

private void OnResponse(object o, ResponseArgs args) { Console.WriteLine (args.ResponseId); }
private void on_buttonDialogoPropio_clicked (object obj, EventArgs args)
{
  Gtk.Window    win    = new Gtk.Window ("Test");
  DialogoPropio dialog = new DialogoPropio(win, Gtk.DialogFlags.DestroyWithParent);
  dialog.Run ();
  dialog.Destroy ();
```

```
  }

  public class DialogoPropio : Dialog
  {
    public DialogoPropio(Gtk.Window w, DialogFlags f) : base("Ttulo", w, f)
    {
      this.Modal = true;
      this.VBox.PackStart( new Label("Etiqueta") );
      this.AddButton("Aceptar", ResponseType.Accept);
      this.AddButton("Cerrar",  ResponseType.Close);
    }

    protected override void OnResponse (ResponseType response_id){
      Console.WriteLine(response_id);
    }
  }
  private void on_app_delete (object o, DeleteEventArgs args) { program.Quit (); }
}
```

Antes de fijarse en los detalles relacionados con los diálogos, debe destacarse cómo se crean los botones de la ventana principal. En el método de creación de la aplicación puede observarse cómo los botones que se añaden se crean con iconos estándar presentes en el sistema. Para referenciarlos, se usa la clase `Stock` de GTK.

Los diálogos se crean en los métodos que se suscriben al evento `Clicked` de cada botón. La estructura de estos métodos suscritos es la misma para todos:

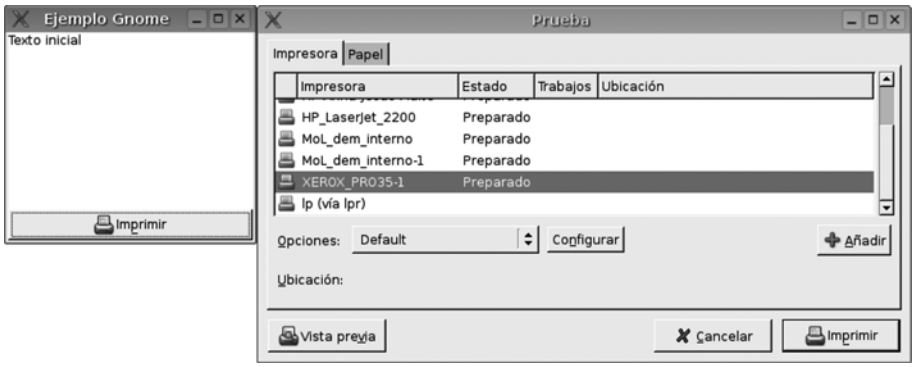
1) creación del diálogo,
2) suscripción al evento `Response`, que se produce en el momento que el usuario finaliza el diálogo,
3) ejecución del diálogo,
4) impresión de los datos recogidos por el diálogo y
5) destrucción del diálogo.

Por último, en el código encontramos la creación del diálogo personalizado propio en forma de clase anidada (`DialogoPropio`) que deriva de la clase `Dialog`. En el constructor de la clase derivada podemos añadir todos los elementos que deseemos que tenga el nuevo diálogo. Por otra parte, el modo de vincular un método al evento `Response` no es usar un delegado, como en los casos anteriores, sino sobrecargar el método `OnResponse`.

6.3. Impresión

El proyecto Gnome, con el objetivo de crear un entorno de escritorio completo, incluye prestaciones como la gestión del sistema de impresión. Una forma sencilla de ofrecer al usuario control sobre este sistema de Gnome es usar el diálogo de impresión que se muestra en la figura 17.

Figura 17



En la imagen puede observarse a la izquierda la ventana de la aplicación de ejemplo, en la que el usuario puede escribir el texto que desea imprimir, y a la derecha el diálogo de impresión estándar de Gnome. En este diálogo, el usuario puede escoger qué impresora desea utilizar, y la configuración de papel. Además, desde él es posible acceder al diálogo de configuración de la impresora o bien a la ventana de previsualización de impresión.

El código siguiente muestra cómo se ha creado esta aplicación ejemplo:

```
using System;
using Gtk;
using Gnome;

class EjemploGnomeImprimir
{
  Program program;
  TextView areaTexto;

  static void Main(string[] args) { new EjemploGnomeImprimir(args);   }

  EjemploGnomeImprimir (string[] args)
  {
    program = new Program("EjemploGnomeImprimir", "0.1", Gnome.Modules.UI , args);

    App app = new App("EjemploGnomeImprimir", "Ejemplo Gnome");
    app.DeleteEvent += new DeleteEventHandler (on_app_delete);

    VBox vb                 = new VBox (false, 0);
    areaTexto               = new TextView ();
    Button buttonImprimir   = new Button (Gtk.Stock.Print);

    areaTexto.Buffer.Text   = "Texto inicial";
    buttonImprimir.Clicked += new EventHandler (on_buttonImprimir_clicked);

    vb.PackStart (areaTexto, true, true, 0);
    vb.PackStart (buttonImprimir, false, true, 0);

    app.Contents = vb;

    app.ShowAll();
    program.Run();
  }

  void ComponerPagina (PrintContext ContextoImp)
  {
    ContextoImp.BeginPage("Demostración");
    ContextoImp.MoveTo(1, 700);
```

```
    ContextoImp.Show(areaTexto.Buffer.Text);
    ContextoImp.ShowPage();
  }

  void on_buttonImprimir_clicked (object o, EventArgs args)
  {
    PrintJob    trabajo   = new PrintJob (PrintConfig.Default ());
    PrintDialog dialogo   = new PrintDialog (trabajo, "Prueba", 0);
    int         respuesta = dialogo.Run ();
    Console.WriteLine ("Respuesta: " + respuesta);

    if (respuesta == (int) PrintButtons.Cancel) {
      Console.WriteLine("Impresión cancelada");
      dialogo.Hide (); dialogo.Dispose (); return;
    }
    PrintContext ctx = trabajo.Context;
    ComponerPagina(ctx);

    trabajo.Close();

    switch (respuesta) {
      case (int) PrintButtons.Print:   trabajo.Print (); break;
      case (int) PrintButtons.Preview:
        new PrintJobPreview(trabajo, "Prueba").Show();
        break;
    }

    dialogo.Hide (); dialogo.Dispose ();
  }

  void on_app_delete (object o, DeleteEventArgs args) { Application.Quit (); }
}
```

Inicialmente se crea la ventana principal de la aplicación en la que se inserta un área de texto y el botón de impresión. A este botón se le suscribe el método `on_buttonImprimir_clicked` al evento `Clicked`. De tal modo que, cuando el usuario pulse el botón, se creará un trabajo de impresión (`PrintJob`) a partir de la configuración por defecto (`PrintConfig.Default`). A continuación, se crea el diálogo de impresión (`PrintDialog`) y se ejecuta.

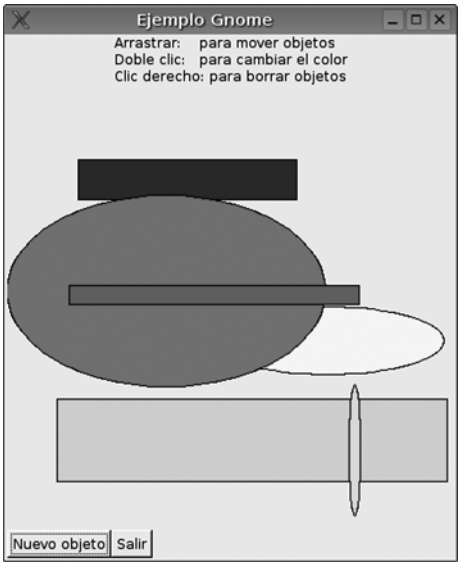
El código retornado por el diálogo (`respuesta`) indica qué acción ha emprendido el usuario. Si el usuario ha pulsado el botón "Cancelar", simplemente oculta el diálogo y después se libera. En caso contrario, se rellena el contenido del trabajo de impresión mediante el método `ComponerPagina` a partir del área de texto. En caso de que el usuario haya pulsado el botón "Imprimir", se imprimirá directamente el trabajo creado. Alternativamente, si el botón pulsado ha sido "Vista previa", se crea una ventana de previsualización de impresión.

6.4. *Canvas*

En caso de que deseemos disponer de un área de dibujo en alguna de las ventanas de la aplicación, podemos usar el elemento `Canvas` de Gnome. Éste permite dibujar fácilmente objetos geométricos básicos y manipular

sus propiedades. A continuación se muestra el código de la aplicación de ejemplo que tiene la interfaz que podéis ver en la figura 18.

Figura 18



Se trata de una ventana principal en la que un conjunto de etiquetas de texto muestran la descripción de los comandos básicos de la aplicación en la parte superior de la ventana. En la parte central se encuentra el elemento gráfico `Canvas`, sobre el que el usuario puede dibujar. Y en la parte inferior, un botón para crear objetos nuevos y otro para salir de la aplicación.

El código que produce y gestiona esta aplicación de ejemplo es el siguiente:

```
using Gnome;
using Gtk;
using Gdk;
using System;

public class EjemploGnomeCanvas {
  Program program;
  private double ini_x = 0.0, ini_y = 0.0;
  private Canvas canvas;
  private Random random = new Random ();
  static void Main(string[] args) { new EjemploGnomeCanvas(args);   }

  EjemploGnomeCanvas (string[] args)
  {
    program = new Program("EjemploGnomeCanvas", "0.1", Gnome.Modules.UI , args);

    App app = new App("EjemploGnomeCanvas", "Ejemplo Gnome");
    app.DeleteEvent += new DeleteEventHandler (on_app_delete);

    VBox vb = new VBox (false, 0);
    vb.PackStart (new Label ("Arrastrar:    para mover objetos\n"   +
            "Doble clic:   para cambia el color\n" +
            "Clic derecho: para borrar objetos"),
        false, false, 0);

    canvas = new Canvas ();
    canvas.SetSizeRequest (400, 400);
    canvas.SetScrollRegion (0.0, 0.0, 400, 400);
```

```
  vb.PackStart (canvas, false, false, 0);

  HBox hb = new HBox (false, 0);

  Button buttonNuevo = new Button ("Objeto nuevo");
  Button buttonSalir = new Button ("Salir");

  buttonNuevo.Clicked += new EventHandler (ObjetoNuevo);
  buttonSalir.Clicked += new EventHandler (Salir);

  vb.PackStart (hb, false, false, 0);
  hb.PackStart (buttonNuevo, false, false, 0);
  hb.PackStart (buttonSalir, false, false, 0);

  app.Contents = vb;
  app.ShowAll();
  program.Run();
}

void ObjetoNuevo (object obj, EventArgs args)
{
  double x1 = random.Next (400), y1 = random.Next (400);
  double x2 = random.Next (400), y2 = random.Next (400);
  double aux;

  if (x1 > x2)        { aux = x1; x1 = x2; x2 = aux; }
  if (y1 > y2)        { aux = y1; y1 = y2; y2 = aux; }
  if ((x2 - x1) < 10) { x2 += 10; }
  if ((y2 - y1) < 10) { y2 += 10; }
  CanvasRE item = null;

  if (random.Next (2) > 0)
    item = new CanvasRect (canvas.Root());
  else
    item = new CanvasEllipse (canvas.Root());
  item.X1 = x1; item.Y1 = y1;
  item.X2 = x2; item.Y2 = y2;
  item.FillColor    = "white";
  item.OutlineColor = "black";
  item.WidthUnits   = 1.0;

  item.CanvasEvent += new Gnome.CanvasEventHandler (on_item_event);
}

void on_item_event (object obj, Gnome.CanvasEventArgs args)
{
  EventButton evento = new EventButton (args.Event.Handle);
  CanvasRE item      = (CanvasRE) obj;
  args.RetVal        = true;

  switch (evento.Type) {
    case EventType.TwoButtonPress: CambiarColor(item);    return;
    case EventType.EnterNotify:    item.WidthUnits = 3.0; return;
    case EventType.LeaveNotify:    item.WidthUnits = 1.0; return;
    case EventType.ButtonPress:
      switch (evento.Button) {
      case 1: ini_x = evento.X; ini_y = evento.Y;   return;
      case 3: item.Destroy();                       return;
      }
      break;
    case EventType.MotionNotify:
      Gdk.ModifierType estado = (Gdk.ModifierType) evento.State;
      if ((estado & Gdk.ModifierType.Button1Mask) != 0) {
        double fin_x = evento.X, fin_y = evento.Y;
        item.Move (fin_x - ini_x, fin_y - ini_y);
        ini_x = fin_x; ini_y = fin_y;
        return;
      }
      break;
```

```
    }

    args.RetVal = false;
    return;
  }

  void CambiarColor (CanvasRE item)
  {
    string[] colors = new string[] {"red", "yellow", "green", "cyan", "blue", "magenta"};
    item.FillColor = colors[random.Next(colors.Length)];
  }

  void Salir         (object obj, EventArgs args)       { Application.Quit (); }
  void on_app_delete (object obj, DeleteEventArgs args) { Application.Quit (); }
}
```

En primer lugar, puede observarse cómo el método constructor crea los elementos comentados en la descripción de la interfaz. Posteriormente, se encuentra el método `ObjetoNuevo`, que ha suscrito al método `Clicked` del botón para añadir nuevos objetos geométricos al área de dibujo. Dicho método se encarga de escoger unas coordenadas aleatorias y una forma geométrica al azar entre rectángulos y elipses. Al crear un ítem de alguna de estas dos formas geométricas, se le pasa a su constructor la referencia del `Canvas` al que debe añadirse. Al mismo tiempo, se le suscribe el método `on_item_event` para que se ejecute en cualquier evento que se produzca sobre dicha forma geométrica. Este método analiza qué evento se ha producido y, en función de él, crea la respuesta:

- **tipo `TwoButtonPress`**: llama al método para cambiar su color (`CambiarColor`).

- **tipo `EnterNotify` /`LeaveNotify`**: modifica el tamaño del borde cuando el ratón está encima de del ítem para indicar que está seleccionado.

- **tipo `ButtonPress`**: si se trata del tercer botón, destruye el ítem. Si, en cambio, se trata del primer botón, se memoriza la posición actual del puntero, como referencia inicial del movimiento del ítem.

- **tipo `MotionNotify`**: si el movimiento se ha producido con el primer botón pulsado, se mueve el ítem siguiendo la relación entre las coordenadas actuales del ratón y las almacenadas anteriormente en el evento tipo `ButtonPress`.

Como puede apreciarse mediante el código de ejemplo, crear y manipular objetos geométricos con la clase `Canvas` de Gnome es senzillo.

6.5. Asistentes

Por último, en este subapartado veremos el uso de asistentes de Gnome. Se trata de un método ágil para crear agrupaciones de diálogos con la finalidad de

llevar a cabo tareas complejas. El esquema básico es un paso inicial informativo, una secuencia de pasos sencillos en los que se va recogiendo la información necesaria y un paso final en el que se pide confirmación para realizar las acciones oportunas en función de la información recogida.

Para ver cómo se crean los asistentes, se ha creado el siguiente ejemplo, que tiene la interfaz que podéis ver en la figura 19.

Figura 19

Consiste en una ventana principal que ofrece el botón mediante el cual se abre el diálogo. El diálogo simplemente consta de los tres pasos mínimos, y recoge un nombre entrado por el usuario, para acabar mostrándolo por la consola de texto. A continuación se muestra el código que lo genera:

```
using System;
using Gtk;
using GtkSharp;
using Gnome;

class EjemploGnomeAsistente
{
  Program program;

  static void Main(string[] args) { new EjemploGnomeAsistente(args);   }

  EjemploGnomeAsistente (string[] args)
  {
    program = new Program("EjemploGnomeAsistente", "0.1", Gnome.Modules.UI , args);

    App app = new App("EjemploGnomeAsistente", "Ejemplo Gnome");
    app.DeleteEvent += new DeleteEventHandler (on_app_delete);

    HBox hb = new HBox();

    Button buttonAsistente = new Button ( Gtk.Stock.Preferences );
    buttonAsistente.Clicked += new EventHandler (on_buttonAsistente_clicked);
    hb.PackStart(buttonAsistente);

    app.Contents = hb;

    app.ShowAll();
    program.Run();
  }
```

```
  private void on_buttonAsistente_clicked (object obj, EventArgs args)
  {
    Druid druid = new Druid();

    DruidPageEdge paso1 = new DruidPageEdge(EdgePosition.Start, true,
        "Título del asistente",
        "Descripción del asistente", null, null, null);

    DruidPageStandard paso2 = new DruidPageStandard("Paso 1",null,null);

    Gtk.Entry nombre = new Gtk.Entry();
    paso2.AppendItem("_Nombre:", nombre, "descripción del ítem");

    DruidPageEdge paso3 = new DruidPageEdge(EdgePosition.Finish, true,
                    "Título de la útima página",
                    "Descripción de la útima página", null, null, null);

    druid.AppendPage(paso1);
    druid.AppendPage(paso2);
    druid.AppendPage(paso3);

    App app = new App("Asistente", "Título de la ventana");

    paso3.FinishClicked += delegate{
      app.Destroy();
      Console.WriteLine("Datos recogidos: {0}", nombre.Text);
    };

    paso1.CancelClicked += delegate { app.Destroy(); };
    paso2.CancelClicked += delegate { app.Destroy(); };
    paso3.CancelClicked += delegate { app.Destroy(); };

    app.Contents = druid;
    app.ShowAll();
  }

  private void on_app_delete (object o, DeleteEventArgs args) { program.Quit (); }
}
```

El método constructor de la aplicación simplemente crea la ventana principal con el botón y lo enlaza al método `on_buttonAsistente_clicked`. Dicho método es el encargado de crear el asistente (`Druid`) con la página de información inicial (`DruidPageEdge` con `EdgePosition.Start`), la página de recogida de información intermedia (`DruidPageStandard`) y la página de confirmación final (`DruidPageEdge` con `EdgePosition.Finish`).

A diferencia de los diálogos previos, para poner en marcha el asistente necesitamos crear una ventana nueva en la que insertarlo como contenido y mostrar la ventana. Finalmente, en el ejemplo se suscriben todos los eventos, desde pulsar los botones de cancelación en cualquiera de los pasos, a métodos anónimos que destruyen el asistente.

El evento que se produce si el usuario llega al último paso del asistente y confirma la operación es `FinishClicked`. En el ejemplo, este método simplemente muestra el nombre por la consola de texto.

7. XML

Extensible Markup Language es hoy en día el formato por excelencia para la transacción de archivos de texto. El uso en la web como moneda de intercambio y la creación de un conjunto de estándares para la gestión y transformación de archivos han hecho de esta arquitectura la más usada en la programación.

Mono soporta la mayoría de los estándares de W3C en torno a XML: XSLT, Xpath, XML Schema, Relax NG y Serialización en XML de los objetos. Aquí vamos a describir brevemente cómo utilizar esta tecnología.

7.1. Leer y escribir XML

Vamos a ver cómo leer y escribir un archivo de disco con una estructura XML. Para poder hacerlo, necesitamos la biblioteca de `System.Xml`, que contiene las clases `XmlReader` y `XmlWriter`.

Escribir en consola:

```
using System.Xml;

....

// Definimos cuál es el esquema que va a utilizarse en el XML
private const string RDF = “http://www.w3.org/1999/02/22-rdf-syntax-ns”;

XmlTextWriter writer = new XmlTextWriter(Console.Out); // vamos a escribir en la consola
writer.Formatting = Formatting.Indented; // vamos a escribir indentando
writer.Indentation = 2; // espacios en la indentación
writer.IdentChar = ‘ ‘; // carácter para la indentación
writer.WriteStartDocument(true); // empezamos a escribir el documento

// Definimos el primer elemento RDF dentro del esquema RDF con el atributo xmlns
// Esto creará:
// <rdf:RDF xmlns=“http://purl.org/rss/1.0/”
//          xmlns:rdf=“http://www.w3.org/1999/02/22-rdf-syntax-ns”>
writer.WriteStartElement(“rdf”,”RDF”,RDF); // comprueba que dentro de RDF hay un elemento rdf
writer.WriteAttributeString(“xmlns”,”http://purl.org/rss/1.0/”);
```

```
// Definimos un elemento nuevo dentro del padre anterior:
// <title>prova</title>
writer.WriteElementString("title","prova");

// Creamos un hijo con un elemento li
// <rdf:li rdf:resource="http://www.tornatmico.org"/>
writer.WriteStartElement("li",RDF);
writer.WriteAttributeString("resource",RDF,"http://www.tornatmico.org");
writer.WriteEndElement();

// Cerramos un elemento
writer.WriteEndElement();

...
```

Para poder leer de un archivo:

```
using System.XML;

...

XmlTextReader reader = new XmlTextReader("archivo.xml");
while (reader.Read()) {
  Console.WriteLine("Tipo de nodo={0}, Nom={1}, Valor=\"{2\"",
                    reader.NodeType, reader.Name, reader.Value);
}
reader.Close();
```

7.2. Navegar y manipular XML en memoria

Las funciones anteriores sirven para poder trabajar de una manera secuencial con los archivos XML. Los podéis leer y escribir, pero sin tener constancia de toda la estructura a la vez. Cuando se escribe, podéis trabajar en el punto donde estáis, y cuando leéis no podéis navegar por el archivo, sólo leerlo secuencialmente. Para poder trabajar en memoria requerimos del uso de un árbol DOM (Document Object Model). Esta estructura nos permite mapear un archivo XML en memoria para poder trabajar con él y navegar de modo no secuencial. Esta manera de usar los XML fue desarrollada por W3C e implementada en todos los lenguajes. Esta es la manera que usan los navegadores para representar internamente el código de las páginas web XHTML.

Para poder leer un archivo en un documento DOM:

```
using System
using System.Xml;
```

```
public class ReadDocument {
  public static void Main (string[] args) {
    string filename = args[0];
    XmlDocument document = new XmlDocument();
    document.Load(fileame);
  }
}
```

Hay dos modos en los que podemos recorrer un árbol XML: en profundidad, siguiendo un orden secuencial, o con un lenguaje llamado Xquery que nos permite escoger una parte del árbol. Para poder buscar en un árbol DOM:

```
 XmlDocument document = new XmlDocument();
  document.Load(filename);
  // Creamos un conjunto de nodos que responden a una xpath query
  XmlNodeList nodes = document.SelectNodes(pregunta);
  foreach (XmlNode node in nodes) {
    ...
  }
```

El problema de usar este modo es que cargamos todo el documento en memoria, lo cual al trabajar con documentos grandes puede suponer una carga extrema. Para esto sirve el tipo `XpathDocument`:

```
// Creamos un documento con referencia a xpath, éste no se carga
en memoria
XPathDocument document = new XpathDocument(filename);
// Creamos un navegador de este documento
XPathNavigator navigator = document.CreateNavigator();
// Creamos un conjunto de nodos que responden a una xpath query
XpathNodeIterator iterator = navigator.Select(query);
while (iterator.MoveNext()) {
  ...
}
```

Al crear un nuevo elemento podemos añadir atributos y decidir a qué punto del árbol XML queremos vincularlo.

```
 // Creamos un elemento
  XmlElement element =
document.CreateElement("prova","contenido");
  // Creamos un atributo
  XmlAttribute attribute =
document.CreateAttribute("info","información");
  // Añadimos un atributo al elemento
  element.Attributes.Append(attribute);
  // Añadimos el elemento a la raíz del documento
  document.DocumentElement.AppendChild(element);
```

7.3. Transformar con XML

Podemos elegir transformar un documento XML con las clases de XSLT. En su ejecución se puede escoger si se usa la biblioteca nativa libxslt usando la variable de entorno `MONO_UNMANAGED_XSLT`. El hecho de usar la biblioteca nativa nos obliga a estar en un entorno `no managed` de modo que el `garbage collector` no actua en los objetos XSLT. Para poder transformar un documento XML:

```
try {
  XslTransform transform = new XslTransform();
  transform.Load(stylesheet);
  transform.Transform(source, target, null);
} catch ( Exception e ) {
  Console.Error.WriteLine(e);
}
```

7.4. Serializar

En Mono se puede serializar un objeto a XML de dos modos, uno con el *runtime* `serialization`, que nos permite generar de un objeto un XML con una sintaxis arbitraria o un formato binario, y XML `serialization`, que nos permite generar SOAP o un XML arbitrario.

Para poder serializar un objeto, debemos poner en la creación que será un objeto serializable, de manera que en el momento de definir las distintas variables podamos decidir si será un `xmlelement` o un `attribute`. Para ver un ejemplo:

```
using System;
using System.Xml;
using System.Xml.Serialization;

// Class main para crear el objeto y serializarlo
public class Persona {
  public static void Main (string[] args) {

    // Creamos el objeto serializable
    Persona obj = new Persona();

    // Creamos un serializador para poder procesar el objeto
   XmlSerializer serializador = new XmlSerializer(obj.GetType());
    // Mostramos por consola el XML devuelto
    serializer.Serialize(Console.Out, obj);
  }
}

// Escogemos que una classe pueda ser serializable
[Serializable]
public class Persona {
  public string Nombre = "Jesus Mariano Perez";
  // Cambiamos el nombre del tag que contendrá la variable
  [XmlElement("VariableXML")]
```

```
  [XmlElement("VariableXML")]
  public int variable = new Random().Next();

  // Esta variable se mostrará como un atributo en el tag del objeto
   [XmlAttribute]
  public DateTime DataActual = DateTime.Now;

  // Creamos un array con dos direcciones para mostrar dos
subelementos
  public Direccion [] midireccion =
    new Direccion [] {
      new Direccion(), new Direccion() };
}

[Serializable]
public class Direccion {

 [XmlAttribute]
  public string Calle = "calle Pepito";

}
```

La salida sería:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<persona DataActual="2006-04-04T10:01:48.245630-05:00">
  <VariableXML>41231253123</VariableXML>
  <Nombre>Jesus Mariano Perez</Nombre>
  <midireccion>
    <direccion Calle="calle pepito"/>
    <direccion Calle="calle pepito"/>

  </midireccion>
</persona>
```

7.5. Restringir XML

Para poder validar un XML podemos cargar un XML Schema y delegar a una función la ejecución de código en caso de error de validación. Para usarlo:

```
...
// Cargamos un archivo para procesarlo
XmlTextReader reader = new XmlTextReader(xmlfile);

// Creamos un validador en el reader
xmlValidatingReader validator = new XmlValidatingReader(reader);

// Queremos validar son XMLSchema
validator.ValidationType=ValidationType.Schema;

// Cargamos el archivo del esquema
validator.Schemas.Add(string.Empty, schemafile);

// Función que se ejecutará cuando detecte un error
validator.ValidationEventHandler += new
ValidationEventHandler(Error);

// Leemos el archivo con el validador
```

```
while (validator.Read()) {

}

....

private static void Error(object sender, ValidationEventArgs e) {
  Console.Errors.WriteLine(e.Message);
}

....
```

8. Biblioteca avanzada

Hay un conjunto de bibliotecas que han sido portadas a Mono usando *bindings* o con implementaciones nativas. Estudiaremos algunas de ellas: una para gestionar archivos, el registro de Gnome; la internacionalización y la compilación automática.

8.1. GnomeVFS

Esta biblioteca nos permite tener una interfaz para poder acceder a archivos estén donde estén. Una vez obtenido el archivo, la biblioteca nos permite saber mediante MIME su tipo de contenido y hacer operaciones sobre éste. Para poder hacer referencia a un archivo debemos usar la URI:

Acceso a archivos

Podemos acceder a archivos de varias maneras. Por ejemplo, podemos acceder mediante un WebDAV, el protocolo HTTP, un sistema de archivos NTFS o el mimo disco duro del sistema.

```
//metodo://usuario:password@direccion/directorio/archivo
http://usuario@passwd@maquina/home/usuario/prueba.tar.gz
```

```
//uri#metodo[/sub_uri]
// Vamos a descomprimir con gzip y con tar para obener un archivo
en el directorio
// /directorio1/archivo
http://usuario@passwd@maquina/home/usuario/
prueba.tar.gz#gzip#tar/directorio1/archivo
```

Podemos monitorizar los cambios, creación y borrado de una URI:

```
// Iniciamos el sistema de GnomeVFS
Gnome.Vfs.Vfs.Initialize ();

// Creamos un monitor
Monitor monitor = new Monitor ();
// Añadimos una función delegada a los cambios
monitor.Changed += OnChanged;
// Añadimos una función delegada al borrado
monitor.Deleted += OnDeleted;
// Añadimos una función delegada a la creación
monitor.Created += OnCreated;
// Añadimos una función al cambio de los metadatos del archivo
monitor.MetadataChanged += OnMetadataChanged;

// Añadimos un directorio a ser monitorizado
monitor.Add ("directorio", MonitorType.Directory);

// Bucle principal de ejecución de eventos
```

```
new MainLoop ().Run ();

// Cerramos el sistema de GnomeVFS
Gnome.Vfs.Vfs.Shutdown ();

public static void OnChanged (string monitor, string uri)
{
         Console.WriteLine ("Uri changed: {0}", uri);
}

public static void OnDeleted (string monitor, string uri)
{
         Console.WriteLine ("Uri deleted: {0}", uri);
}

public static void OnCreated (string monitor, string uri)
{
          Console.WriteLine ("Uri created: {0}", uri);
}

public static void OnMetadataChanged (string monitor, string uri)
{
          Console.WriteLine ("Uri metadata changed: {0}", uri);
}
```

Podemos saber el tipo de contenido en un archivo procedente de una URI:

```
Gnome.Vfs.Vfs.Initialize ();
// Creamos un tipo de objecto uri
Gnome.Vfs.Uri uri = new Gnome.Vfs.Uri ("/tmp/archivo");
// Analizamos el tipo de archivo que es
MimeType mimetype = uri.MimeType;
// Mostramos el tipo de archivo que es
Console.WriteLine ("La uri `{0}' parece ", uri, mimetype.Name);
Gnome.Vfs.Vfs.Shutdown ();
```

Podemos hacer operaciones (crear, modificar y borrar) sincronizadas:

```
Gnome.Vfs.Vfs.Initialize ();

Gnome.Vfs.Uri uri = new Gnome.Vfs.Uri ("/tmp/prueba");

// Abrimos el archivo en modo escritura
Handle handle = Sync.Open (uri, OpenMode.Write);

// Escogemos el tipo de encoding utf8 para grabar el archivo
UTF8Encoding utf8 = new UTF8Encoding ();

Result result = Result.Ok;
Console.WriteLine ("Enter text and end with Ctrl-D");
while (result == Result.Ok) {
   // Mientras haya alguna línea que leer y el resultado de grabar
sera correcto, leer de consola.
    string line = Console.ReadLine ();
    if (line == null)
        break;
    byte[] buffer = utf8.GetBytes (line);

    ulong bytesWritten;
   // Escribimos en el archivo. El sistema se bloqueará hasta que
acabe la escritura
```

```
    result = Sync.Write (handle, out buffer[0],
        (ulong)buffer.Length, out bytesWritten);
    Console.WriteLine ("resultado escritura '{0}' = {1}", uri,
result);
    Console.WriteLine ("{0} bytes escritos", bytesWritten);
}

// Cerramos el archivo
result = Sync.Close (handle);
Console.WriteLine ("resultado cerrar '{0}' = {1}", uri, result);

Gnome.Vfs.Vfs.Shutdown ();
```

Podemos hacer operaciones asíncronas:

```
// De este modo podemos abrir, leer, escribir, borrar y cerrar archivos de forma que no bloquee
// el programa principal

using GLib;
using Gnome.Vfs;
using System;
using System.Text;
using System.Threading;
namespace TestGnomeVfs {

        public class TestAsync {
                private static MainLoop loop;
                private static Handle handle;

                static void Main (string[] args)
                {
                        // Iniciamos el sistema de GnomeVFS
                        Gnome.Vfs.Vfs.Initialize ();

                        // Creamos un objeto uri de la cadena entrada por consola
                        Gnome.Vfs.Uri uri = new Gnome.Vfs.Uri (args[0]);
                        // Creamos un thread para que abra un archivo para leer
                        // con la prioridad por defecto.
                        // Una vez que ha sido abierto, llamará a la operación OnOpen
                        handle = Async.Open (uri, OpenMode.Read,
                                             (int)Async.Priority.Default,
                                             new Gnome.Vfs.AsyncCallback (OnOpen));

                        // Empezamos la ejecución del bucle principal de eventos
                        loop = new MainLoop ();
                        loop.Run ();

                        Gnome.Vfs.Vfs.Shutdown ();
                }

                private static void OnOpen (Handle handle, Result result)
                {
                        // Cuando se ha abierto el archivo se llamará está función
                        Console.WriteLine ("Uri opened: {0}", result);
                        if (result != Result.Ok) {
                                // En caso que haya ido mal, cerrará el bucle
                                // principal de eventos
                                loop.Quit ();
                                return;
                        }
                        // Definimos un buffer para poder leer el archivo
                        byte[] buffer = new byte[1024];
                        // Creamos un thread para que lea el archivo
                        // y lo guarde en el buffer.
```

```
                    // Una vez leído, llamará la función OnRead
                      Async.Read (handle, out buffer[0], (uint)buffer.Length,
                                  new AsyncReadCallback (OnRead));
                }

                private static void OnRead (Handle handle, Result result,
                                            byte[] buffer, ulong bytes_requested,
                                            ulong bytes_read)
                {
                        // Una vez leído parte del archivo
                        // ( el tamaño del buffer ) se ejecuta está función
                        Console.WriteLine ("Read: {0}", result);
                        if (result != Result.Ok && result != Result.ErrorEof) {
                                // Si ha ocurrido un error salimos del bucle principal
                                loop.Quit ();
                                return;
                        }
                        // Decidimos en qué codificación vamos a trabajar
                        UTF8Encoding utf8 = new UTF8Encoding ();
                        // Escribimos por pantalla el contenido del buffer
                        Console.WriteLine ("read ({0} bytes): '{1}'", bytes_read,
                                           utf8.GetString (buffer, 0,
                                           (int)bytes_read));
                        // Si no hemos acabado de leer el archivo ( bytes_read=0)
                        // creamos otro thread
                        // para que siga leiendo y cuando llene el buffer de nuevo
                        // vuelva a llamar
                        // a la función OnRead
                        if (bytes_read != 0)
                                Async.Read (handle, out buffer[0], (uint)buffer.Length,
                                            new AsyncReadCallback (OnRead));
                        else
                                // En caso de terminar el archivo, lo cerramos de
                                // forma asíncrona
                                Async.Close (handle,
                                             new Gnome.Vfs.AsyncCallback (OnClose));
                }
                private static void OnClose (Handle handle, Result result)
                {
                        // Cuando el thread creado para cerrar el archivo lo cierra,
                        // llama a esta función
                        Console.WriteLine ("Close: {0}", result);
                        loop.Quit ();
                }
        }
}
```

8.2. Gconf

Para guardar los parámetros personalizados de un programa podemos hacerlo guardando un conjunto de variables en el registro de Gnome (gconf). Este sistema tiene forma de árbol, donde guarda todas las configuraciones de cada programa y del escritorio. En este arbol existe una definición de su estructura llamada GConf Schema. De esta modo nos permite restringir su contenido.

A continuación tenéis un ejemplo para guardar qué archivo se usa para hacer *log* y si se activa o no:

```
using GConf;
```

```
...
GConf.Client client;
// Base donde vamos a guardar las dos variables
static string KEY_BASE="/apps/monodemo";
// Variable de texto donde guardamos el nombre del archivo
static string KEY_FILENAME = KEY_BASE + "/archivo";
// Variable booleana donde guardamos si hacemos log o no
static string KEY_LOGGING = KEY_BASE + "/log";

// Creamos un cliente del servicio gconf
client = new Gconf.Client();
// Añadimos un delegate a gconf para que, en caso de que se
modifique cualquier variable
// a partir de KEY_BASE llame la función GConf_Changed
client.AddNotify(KEY_BASE,new NotifyEventHandler(GConf_Changed));
...

// Para obtener los valores
try {
  NombreTexto = (string) client.Get(KEY_FILENAME);
  Activo = (bool) client.Get(KEY_LOGGING);
} catch (GConf.NoSuchKeyException e) {
  // No existe la key
} catch (System.InvalidCastException e) {
  // No es del tipo solicitado
}

...

// Para guardar los valores
client.set(KEY_LOGGING, true);
client.set(KEY_FILENAME,"/tmp/prueba.log");
```

8.3. Internacionalización

Hoy en día, el desarrollo de aplicaciones contempla en la mayoría de los casos su traducción. Aunque el público objetivo de una aplicación sea inicialmente un colectivo con una lengua determinada, es previsible que a medio plazo se desee distribuir el programa entre colectivos que usan otras lenguas.

Una buena práctica es desarrollar las aplicaciones en inglés como idioma básico y crear una primera versión traducida al castellano. Si seguimos esta metodología, aseguraremos dos puntos importantes:

1) la aplicación desarrollada está totalmente preparada para ser traducida a otros idiomas, y

2) la aplicación ya está disponible en inglés, de modo que pueden acceder para hacer sus traducciones otras personas que hablen diferentes idiomas.

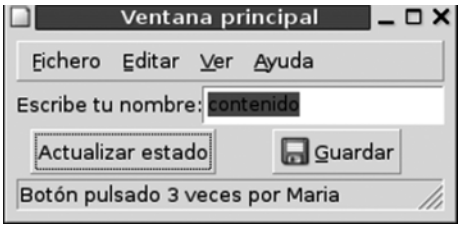
Normalmente, la traducción de los textos de una aplicación es solamente una parte del proceso de **internacionalización** y, aparte de éstos, también debe te-

Abreviaremos *internacionalización* con "i18n".

nerse en cuenta que en las diferentes culturas existen formatos diferentes para fechas, cantidades numéricas, día inicial de la semana, etc.

Para mostrar cómo se pueden traducir fácilmente las aplicaciones que realicemos usando la arquitectura expuesta en este módulo (C#+GTK+Glade), usaremos un ejemplo. Se trata de una aplicación no funcional, simplemente con ánimo de mostrar los diferentes contextos en los que podemos encontrar cadenas de texto que deben traducirse. La aplicación tiene la interfaz creada con Glade que podéis ver en la figura 20.

Figura 20



Se trata de una interfaz que ofrece al usuario un menú convencional, un área de contenido con una etiqueta, un campo de texto y dos botones; y una barra de estado. La funcionalidad es mínima, y consiste en actualizar el texto de la barra de estado en el momento en que el usuario pulsa el botón izquierdo. La barra de estado indicará el número de veces que el usuario con el nombre indicado en el campo de texto ha pulsado el botón izquierdo.

En primer lugar vemos el código de la aplicación, que ha sido desarrollada de entrada en inglés:

```
using Gtk;
using System;
using Glade;
using Mono.Unix;

public class EjemploI18Ngui
{
  [Widget] Statusbar barraEstado;
  [Widget] Entry     entryNombre;
  int                cont;

  public static void Main (string[] args)
  {
    Catalog.Init("DominioTraduccion","./locale");
    Application.Init ();
    new EjemploI18Ngui();
    Application.Run();
  }

  public EjemploI18Ngui()
  {
    XML glade = new Glade.XML("EjemploI18Ngui.glade",
                              "window1",
                              "DominioTraduccion");
    glade.Autoconnect (this);
    cont = 0;
    barraEstado.Push( 0, Catalog.GetString("Welcome!") );
```

```
  }

  void on_botonBEstado_clicked(object o, EventArgs args)
  {
    cont++;

    string formato = Catalog.GetPluralString(
                       "{0} pushed the button once",
                                "{0} pushed the button {1} times",
                                   cont);

    string mensaje = String.Format(formato, entryNombre.Text, cont);

    barraEstado.Pop(  0 );
    barraEstado.Push( 0, mensaje);
  }

  void on_my_menu_entry1_activate   (object o, EventArgs args){}
  void on_new1_activate             (object o, EventArgs args){}
  void on_open1_activate            (object o, EventArgs args){}
  void on_save1_activate            (object o, EventArgs args){}
  void on_save_as1_activate         (object o, EventArgs args){}
  void on_quit1_activate            (object o, EventArgs args){}
  void on_cut1_activate             (object o, EventArgs args){}
  void on_copy1_activate            (object o, EventArgs args){}
  void on_paste1_activate           (object o, EventArgs args){}
  void on_delete1_activate          (object o, EventArgs args){}
  void on_about1_activate           (object o, EventArgs args){}

  private void OnWindowDeleteEvent (object o, DeleteEventArgs args)
  {
    Application.Quit ();
    args.RetVal = true;
  }
}
```

La compilación de este ejemplo debe realizarse incluyendo `Mono.Posix` para disponer de las herramientas de traducción, aparte de los recursos habituales para trabajar con Glade:

```
mcs -pkg:Gtk-sharp,glade-sharp -r:Mono.Posix -
resource:EjemploI18Ngui.glade EjemploI18Ngui.cs
```

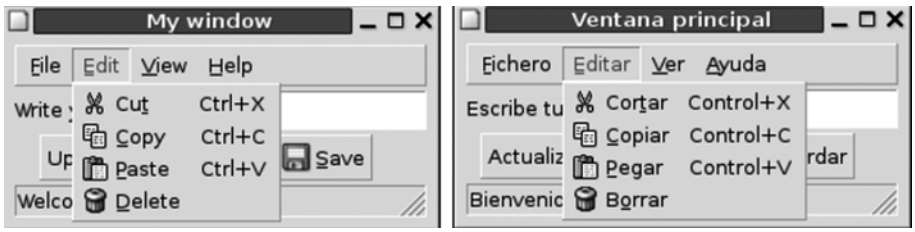
En primer lugar, vemos como el método principal indica cuál es el nombre del dominio de traducción (`DominioTraduccion`) y el directorio donde se podrán encontrar los archivos de traducción de las cadenas de texto. Más adelante ampliaremos este punto.

1) Elementos estándares del sistema traducidos directamente

A continuación, se llama al constructor de la aplicación que creará la interfaz a partir del archivo de descripción Glade en XML. Notad que en esta ocasión el último parámetro de `Glade.XML` no es `null`, como en los ejemplos de los apartados anteriores, sino el dominio de traducción. Al indicar este dominio de traducción, el creador de la interfaz no solo creará los elementos definidos en el archivo XML, sino que para todos aquellos que sean elementos estándar del sistema buscará automáticamente su traducción.

Los elementos estándar del sistema son, por ejemplo, los botones de acciones predeterminadas (en este caso, el botón "Guardar" y las opciones internas de los menús (figura 21).

Figura 21



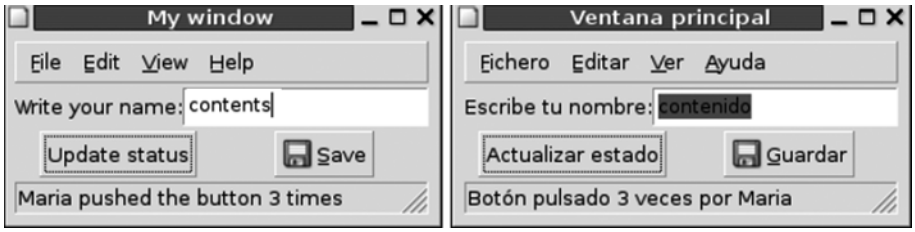
En la figura 21 se ve la misma aplicación ejecutada usando el idioma inglés y el idioma español. En ambos casos las etiquetas que aparecen en las entradas del menú de edición y el botón para guardar se traducen automáticamente gracias a la traducción de elementos estándar que normalmente se encuentra en:

```
/usr/share/locale/<idioma>/LC_MESSAGES/<dominioTraduccion>.mo
```

2) Catalog.GetString

Una vez creada la interfaz, se inicializa la barra de estado con el mensaje de bienvenida. Para crear el texto que insertaremos en la barra de estado, se llama al método `GetString` del catálogo de traducciones (clase `Catalog` de `Mono.Unix`). Este método busca en el catálogo del dominio indicado en el método `Main` y retorna su traducción al idioma del entorno de ejecución.

Figura 22



El idioma del contexto de ejecución puede cambiarse mediante las variables de entorno. Las dos imágenes anteriores han sido creadas con las siguientes líneas de ejecución:

```
LANGUAGE=en mono EjemploI18Ngui.exe
LANGUAGE=es mono EjemploI18Ngui.exe
```

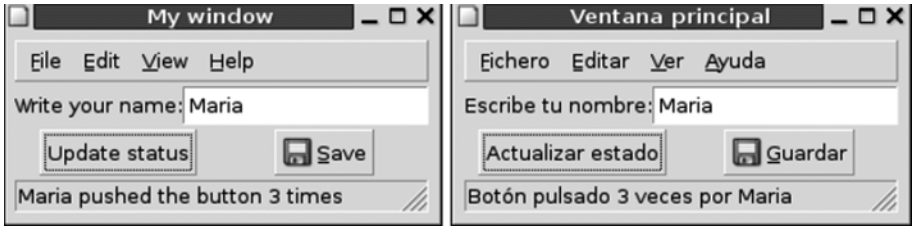
El resto del código simplemente corresponde a los métodos suscritos a los diferentes eventos de la interfaz, tanto los ocasionados por los botones como los ocasionados por las opciones de menú. El único método que se ha implemen-

tado internamente es el que da respuesta al botón de actualizar la barra de estado. Inicialmente se asigna un mensaje de bienvenida a la barra de estado, pero cuando el usuario pulsa el botón de actualización se modifica el mensaje para que indique el número de veces que el usuario ha pulsado el botón. Para conseguir un mensaje para el singular diferente del el plural, existe el método Catalog.GetPluralString, que escoge entre dos mensajes en función del último parámetro numérico. Si es 1 escoge el primer mensaje, y si es mayor que 1 escoge el segundo mensaje:

```
string formato = Catalog.GetPluralString(
                                    "{0} pushed the button once",
                                "{0} pushed the button {1} times",
                                    cont);
```

Con estos dos métodos de Catalog podemos realizar manualmente la traducción de cualquier elemento que no sea estándar del stock de Gnome.

Figura 23



3) Generación de los archivos de traducción

Para generar los archivos de traducción es necesario extraer en primer lugar las cadenas susceptibles de traducción del código original. Para hacerlo, usaremos el comando xgettext:

```
xgettext --join-existing --from-code=UTF-8 EjemploI18Ngui.cs
EjemploI18Ngui.glade -o EjemploI18Ngui-es.po
```

Al comando xgettext le indicaremos qué archivos deseamos explorar en busca de cadenas traducibles (tanto archivos de código fuente en C# como de descripción de interfaz Glade) y la codificación de caracteres usada. El resultado es un archivo con extensión .po que contiene todas las cadenas que pueden traducirse. Este archivo lo podemos modificar con cualquier editor de texto o bien usando alguna herramienta dedicada a ello. El resultado de la traducción al español para el ejemplo visto es el siguiente:

```
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2006-06-21 19:17+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
```

```
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=ISO-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"

#: EjemploI18Ngui.glade:8
msgid "My window"
msgstr "Ventana principal"

#: EjemploI18Ngui.glade:35
msgid "_File"
msgstr "_Archivo"

#: EjemploI18Ngui.glade:99
msgid "_Edit"
msgstr "_Editar"

#: EjemploI18Ngui.glade:148
msgid "_View"
msgstr "_Ver"

#: EjemploI18Ngui.glade:157
msgid "My menu entry"
msgstr "Opción personalizada"

#: EjemploI18Ngui.glade:170
msgid "_Help"
msgstr "_Ayuda"

#: EjemploI18Ngui.glade:179
msgid "_About"
msgstr "A_cerca de..."

#: EjemploI18Ngui.glade:218
msgid "Write your name:"
msgstr "Escribe tu nombre:"

#: EjemploI18Ngui.glade:243
msgid "contents"
msgstr "contenido"

#: EjemploI18Ngui.glade:272
msgid "Update status"
msgstr "Actualizar estado"

#: EjemploI18Ngui.cs:45
#, csharp-format
msgid "{0} pushed the button once"
msgid_plural "{0} pushed the button {1} times"
msgstr[0] "Botón pulsado una vez por {0}"
msgstr[1] "Botón pulsado {1} veces por {0}"

#: EjemploI18Ngui.cs:37
msgid "Welcome!"
msgstr "Bienvenido!"
```

A partir del archivo .po generaremos el archivo .mo para el idioma en el que hemos hecho la traducción, usando el comando msgfmt:

```
msgfmt EjemploI18Ngui-es.po -o locale/es/LC_MESSAGES/
DominioTraduccion.mo
```

El archivo .mo tendrá el nombre del dominio indicado al llamar al comando Catalog.Init, y estará colocado en el directorio que hemos indicado con dicha inicialización, o bien en el directorio compartido de traducciones del sistema (por ejemplo, /usr/share/locale/es/LC_MESSAGES/).

8.4. Nant y Nunit

8.4.1. Nant

Nant es un sistema para compilar todo un proyecto usando la filosofía de Ant en Java. Construyendo un archivo `.build` podemos definir las distintas normas para poder testear, compilar, ejecutar o instalar fácilmente una aplicación. Este archivo funciona al estilo del típico `Makefile`, de modo que podemos definir normas que requieren otras normas y podemos delegar a otros archivos las distintas operaciones. Nos permite trabajar con conjuntos de archivos dentro de un zip, un tar, en un cvs, empotrados en el *assembly* o en un directorio. Las funciones básicas son:

- **csc:** Compilar un archivo `.cs`. En esta tarea se le puede pedir que el objetivo sea una biblioteca o un ejecutable, que tenga distintos recursos referenciados o encastados dentro, que omita errores de *warning* de valores concretos, que use un conjunto de paquetes, y que use un conjunto amplio de archivos de código fuente. Ejecuta distintos compiladores dependiendo de si usamos .NET framework o Mono.

- **copy/delete/move:** Nos permite trabajar con los archivos.

- **exec:** Ejecuta un programa externo.

Para poder hacer una compilación básica, aquí tenemos un ejemplo de archivo `.build`:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project name="proyecto1" default="build" basedir=".">
        <description>Proyecto de prueba</description>
        <property name="project.name" value="Proyecto1"/>
        <property name="project.version" value="2.0"/>
        <description>Ponemos el debug a cierto como variable global</description>
        <property name="debug" value="true"/>
        <target name="clean" description="Limpiar">
                <echo message="Limpiando"/>
                <delete>
                      <description>Vamos a borrar un archivo en el directorio bin</description>
                        <fileset basedir="./bin/">
                        <include name="Programa.exe"/>
                        <include name="Biblioteca.dll"/>
                        </fileset>
                </delete>
        </target>
        <description>Creamos una regla para compilar el programa. Requiere la
```

```
            biblioteca para poder compilarla</description>
    <target name="programa" description="Programa" depends="biblioteca">
        <echo message="Compilando programa"/>
        <csc target="exe" output="bin/Programa.exe" debug="${debug}">
            <sources failonempty="true">
                <include name="src/Programa.cs"/>
            </sources>
            <references>
                <include name="bin/Biblioteca.dll"/>
            </references>
        </csc>
    </target>
    <description>Creamos una regla para compilar la biblioteca</description>
    <target name="biblioteca" description="Biblioteca">
        <echo message="Compilando biblioteca"/>
        <csc target="library" output="bin/Biblioteca.dll" debug="${debug}">
            <sources failonempty="true">
                <include name="src/libPrograma/*.cs"/>
                <include name="src/archivobiblio.cs"/>
                <include name="src/libPrograma2/*.cs"/>
            </sources>
            <description>Las referencias de dlls </description>
            <references>
                <include name="System.Data.dll"/>
                <include name="System.Xml.dll"/>
                <include name="Npgsql.dll"/>
            </references>
            <description>Recursos que se añadirán al binario</description>
            <resources>
                <include name="config.xml"/>
            </resources>
        </csc>
    </target>
```

Para poder compilarlo ejecutaríamos:

```
nant clean; // limpiamos los directorios
nant biblioteca; // compilamos la biblioteca
nant programa; // compilamos el programa
```

8.4.2. Nunit

Nunit es un sistema para el desarrollo de aplicaciones con la metodología orientada a pruebas. Esta biblioteca nos permite programar distintas clases que nos permitirán testear el funcionamiento de nuestras aplicaciones de un modo sistemático. Con este método podemos primero desarrollar las pruebas y luego programar la aplicación para que cumpla los distintos tests.

Un test no es más que una clase que usa las funciones de `assert` para comprobar que una variable tiene un valor en concreto después de usar las funciones programadas y nos permite verificar si se lanzan excepciones u otras comprobaciones.

Podemos hacer *asserts* para comprobar la igualdad de dos decimales o enteros, de dos reales con tolerancia y objetos. Podemos comprobar si dos objetos son

el mismo, si un objeto está dentro de una lista, comprobar si enteros, decimales o reales son mayores o menores, si un objeto es una instancia de un tipo, si es nulo, no nulo, cierto o falso y si una cadena contiene un valor.

Creamos un test de una clase que queramos desarrollar:

```
// Creamos un namespace con la clase que queremos desarrollar y los
tests
namespace banco
{
  using System;
  using NUnit.Framework;

  // Queremos comprobar el funcionamiento de una clase que gestiona
las cuentas de un banco

  [TestFixture]
  public class CuentaTest
  {
    Cuenta origen;
    Cuenta destino;

    // Configuración de los tests
    [SetUp]
    public void Init()
    {
      // Creamos una cuenta de prueba
      origen = new Cuenta();
      // Añadimos una cantidad
      origen.Deposit(200.00F);
      // Creamos una cuenta de prueba
      destinacion = new Cuenta();
      // Añadimos una cantidad
      destinacion.Deposit(150.00F);
    }

    // Así definimos un test
    [Test]
    public void TransferFunds()
    {
     // Transferimos una cantidad de dinero desde el origen al destino
      origen.TransferFunds(destination, 100.00f);
      // Con este test comprobamos que el destino tenga 250
      Assert.AreEqual(250.00F, destinacion.Balance);
      // Y también comprobamos si al origen le queda 100
      Assert.AreEqual(100.00F, origen.Balance);
    }

     // Test en el que esperamos recibir una excepción
    [Test]
    // Esperamos esta excepción
    [ExpectedException(typeof(NoHayFondosException))]
    public void TransferWithInsufficientFunds()
    {
      origen.TransferFunds(destinacion, 300.00F);
    }

    // Test que no queremos que se ejecute
    [Test]
    [Ignore("Ignoramos este test")]
    public void TransferWithInsufficientFundsAtomicity()
    {
      try
      {
        origen.TransferFunds(destinacion, 300.00F);
      }
```

```
      catch(NoHayFondosException expected)
       {
       }

       Assert.AreEqual(200.00F,origen.Balance);
       Assert.AreEqual(150.00F,destinacion.Balance);
    }
  }
}
```

Una vez que tenemos el test, podemos realizar la clase:

```
namespace banco
{
  using System;

  // Excepción que se usará cuando no hay fondos
  public class NoHayFondosException: ApplicationException
  {
  }

  public class Cuenta
  {
    // Variable del balance
    private float balance;

    // Variable que no se puede modificar desde el exterior con el balance mínimo
    private float minimumBalance = 10.00F;
    public float MinimumBalance
    {
       get{ return minimumBalance;}
    }

    // Añadimos dinero a la cuenta
    public void Deposit(float amount)
    {
      balance+=amount;
    }

    // Quitamos dinero a la cuenta
    public void Withdraw(float amount)
    {
      balance-=amount;
    }

    // Enviamos dinero de una cuenta a otra
    public void TransferFunds(Account destination, float amount)
    {
      // Si el balance menos la cantidad es menor que el mínimo, lanzamos una excepción
      if(balance-amount<minimumBalance)
        throw new NoHayFondosException();
      // Si no, añadimos y sacamos el dinero
      destination.Deposit(amount);
      Withdraw(amount);
    }

    public float Balance
    {
      get{ return balance;}
    }
  }
}
```

Una vez que tenemos los dos códigos, podemos compilarlos con:

```
mcs -pkg:nunit testCuenta.cs Cuenta.cs -target:library -
out:Cuenta.lib
```

Una vez que tenemos el ensamblado, podemos comprobar los test con la herramienta de consola:

```
nunit-console Cuenta.lib
```

También podemos usar las aplicaciones gráficas gnunit y monodevelop para que nos muestren gráficamente el estado de todos los tests.

9. Aplicaciones

Este último apartado del módulo de creación de aplicaciones con interfaz gráfico en Gnome repasa algunas de las primeras aplicaciones de escritorio desarrolladas con la tecnología expuesta: Mono y GTK. Se trata de aplicaciones de escritorio que ponen de manifiesto el alto rendimiento de la tecnología Mono, ya que efectúan tareas sobre gran cantidad de datos de modo muy eficiente. A pesar de su gran capacidad y prestaciones, todas las aplicaciones tienen una interfaz lo más simplificada posible, siguiendo las líneas de usabilidad ya comentadas de Gnome.

Las cuatro aplicaciones descritas son las siguientes:

- **F-Spot**: gestor de fotografías, diseñado para manejar grandes colecciones. Ofrece opciones de clasificación y exportación de éstas.

- **MCatalog**: catálogo de discos, libros y películas. Ofrece múltiples formas de búsqueda e importación/exportación de datos.

- **Muine**: reproductor de música, ideado para explorar colecciones grandes de archivos musicales. Ofrece descarga de carátulas e información básica sobre las pistas reproducidas.

- **Beagle**: buscador de datos en nuestros medios de almacenamiento locales. Ofrece tiempos de respuesta mínimos para localizar cualquier tipo de información relacionada con las palabras clave.

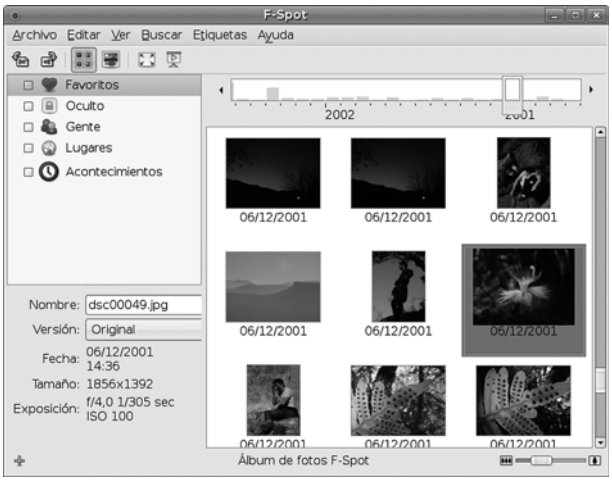
9.1. F-Spot

La aplicación de escritorio F-Spot permite gestionar gran volumen de fotografías mediante una interfaz muy sencilla. Sus capacidades van más allá de la clasificación y localización de fotografías, y llegan hasta funcionalidades básicas de edición de imagen y publicación.

La interfaz básica se muestra en la figura 24.

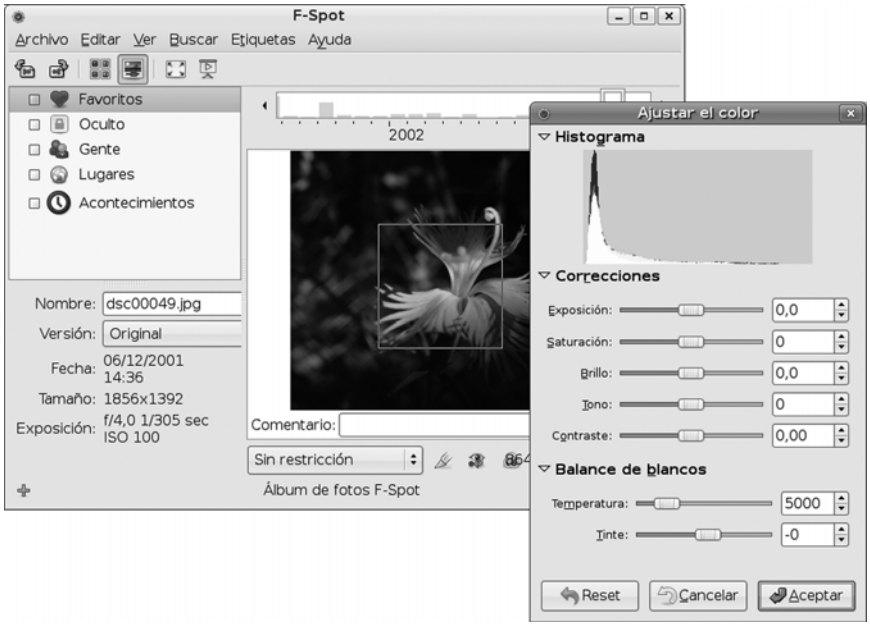
Se trata de una ventana dividida en cuatro áreas principales: información sobre la imagen escogida (inferior izquierda), categorías (superior izquierda), barra de desplazamiento del contenido (superior derecha) y área de contenido (inferior derecha). La barra de desplazamiento del contenido puede basarse tanto en la fecha de captura de las imágenes como en las carpetas que las contienen.

Figura 24



El área de contenido muestra por defecto el conjunto de miniaturas de las fotografías correspondientes al período seleccionado y las categorías seleccionadas. El tamaño de las miniaturas puede graduarse fácilmente con la barra de escala inferior. Al hacer doble clic sobre una imagen, ésta se muestra ampliada ocupando toda el área de contenido. En el momento en que se muestra ampliada una de las imágenes, es posible realizar operaciones de manipulación básicas sobre ésta. Estas operaciones incluyen la reducción de ojos rojos, reenfoque, cambio a blanco y negro, cambio a tonos sepia y ajustes del color, como muestra la figura 25.

Figura 25



F-Spot guarda toda la metainformación referente a las imágenes en una base de datos propia. Cuando realizamos las operaciones de manipulación básica descritas anteriormente, también guarda automáticamente una copia de la imagen original. De tal modo que en cualquier momento es posible ver las versiones anteriores de la misma imagen.

Otro punto destacado de la aplicación es la importación y exportación de imágenes. Por ejemplo, en la figura 26 se aprecia cómo está importando más de diez mil fotografías.

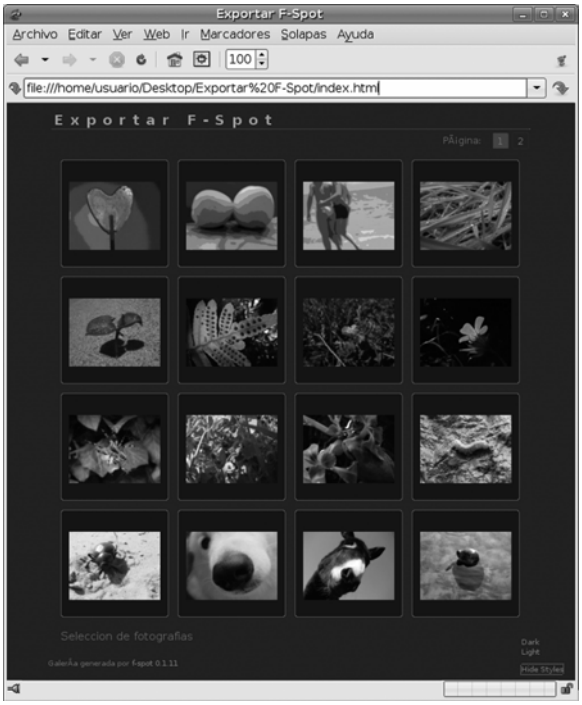
Figura 26

Al mismo tiempo, es posible exportar cualquier selección de las fotografías gestionadas por F-Spot a distintos destinos, como por ejemplo la web Flickr o cualquier web con el sistema Gallery, así como carpetas locales (figura 27).

Figura 27

La exportación a carpetas locales permite dejar sólo los archivos de las imágenes, o crear unas páginas web que las contengan (figura 28).

Figura 28



El desarrollo de la aplicación F-Spot ha resultado interesante para poner en práctica la conexión entre el código C# y las bibliotecas ya existentes de manipulación de imagen. El resultado es un rendimiento muy alto que no se ve penalizado por tener la interfaz creada con un lenguaje basado en máquina virtual.

9.2. MCatalog

La aplicación MCatalog está diseñada para gestionar colecciones de libros, música o películas. Su interfaz intuitiva imita la disposición de los volúmenes clasificados en una biblioteca virtual, de forma que su exploración visual recuerda a la búsqueda de volúmenes en una biblioteca tradicional.

La interfaz básica de la aplicación muestra cuatro áreas principales (figura 29): la lista de colecciones (izquierda), el contenido de la colección (centro), la descripción del elemento seleccionado (derecha), y la barra de búsqueda (inferior).

Figura 29

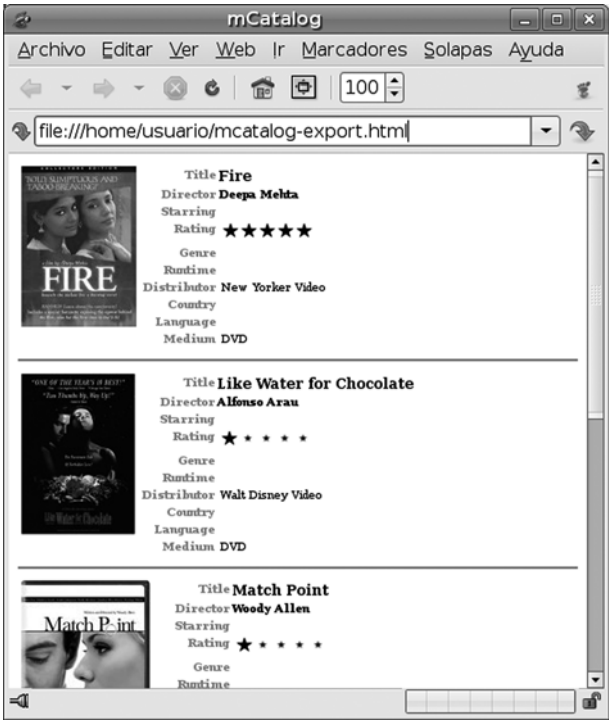
Siguiendo las líneas de diseño de Gnome, su uso resulta sencillo y intuitivo. En el momento en que deseemos añadir un volumen, tenemos la posibilidad de indicar algunas palabras clave y realizar una búsqueda por Internet para no tener que rellenar el resto de los campos (figura 30).

Figura 30

Automáticamente se localizan los metadatos, incluida la carátula del volumen, que será mostrada en la vista virtual de la biblioteca.

Aparte de las posibilidades de gestión internas de la información, que se almacena en una base de datos local SQLlite, también es posible exportar la colección a otros formatos. Por ejemplo a una página web, tal como muestra la figura 31.

Figura 31



La aplicación Mcatalog resulta interesante para comprobar el buen rendimiento de la interacción de Mono con las bases de datos.

9.3. Muine

Muine es un reproductor de música con capacidad para gestionar grandes colecciones. Su simple interfaz proporciona una forma sencilla de acceder a cualquier canción o grupo de canciones que esté en nuestra colección.

La interfaz principal muestra tres áreas principales, tal como muestra la figura 32.

La barra de control (superior), el área de información sobre la canción reproducida (intermedia) y la lista de reproducción (inferior). La lista de reproducción puede alterarse fácilmente para añadir, borrar, reordenar y buscar títulos de forma individual. Aparte, mediante el botón “Reproducir álbum” de la barra de control, es posible seleccionar todas las canciones pertenecientes a un mismo álbum (figura 33).

Figura 32

Figura 33

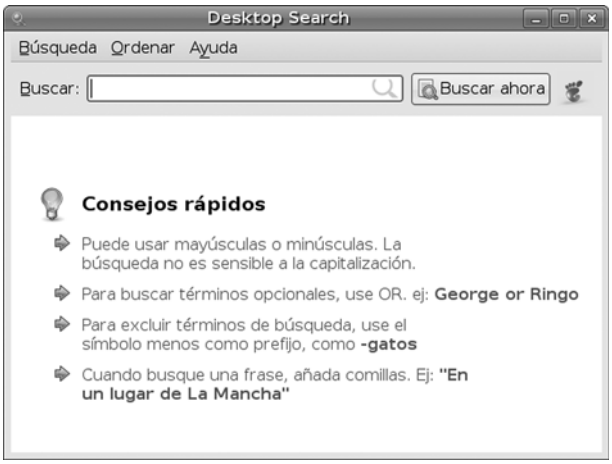
Muine localiza automáticamente por Internet la carátula de los álbumes, que muestra en el momento en que los estamos explorando o cuando se reproduce una canción que está incluida en ellos.

El desarrollo de la aplicación Muine ha ido acompañado desde sus inicios de la mejora de los vínculos entre la arquitectura Mono y el sistema de reproducción multimedia de Gnome gstreamer.

9.4. Beagle

Por último, exponemos la aplicación de búsqueda de contenido en el ordenador de escritorio que más sorpresa ha levantado por su buen rendimiento: Beagle. Su interfaz es minimalista, al estilo del buscador Google o su herramienta equivalente Google Desktop (figura 34).

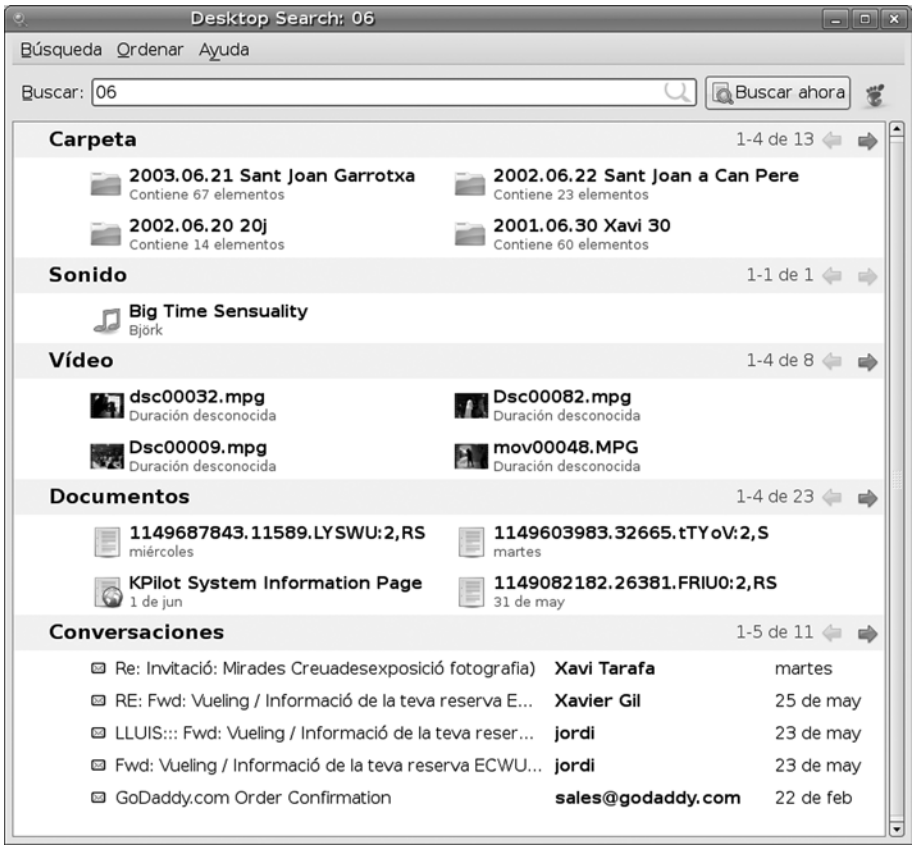
Figura 34



Se muestra una caja de texto en la que introducir las palabras clave de la búsqueda, que admite la sintaxis habitual para expresar criterios compuestos. Esta interfaz es independiente del motor que crea el índice de los datos en disco y que se ejecuta en paralelo y se actualiza a medida que se producen cambios.

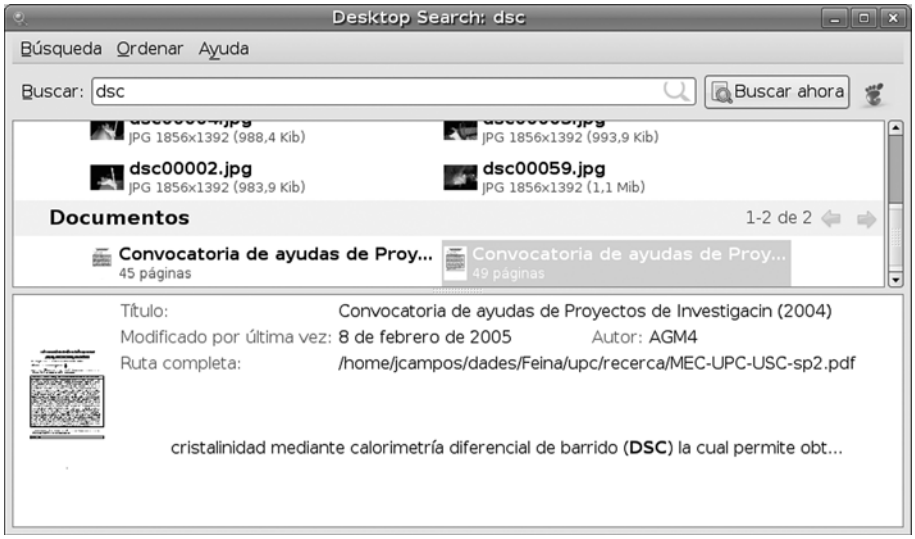
Los elementos que puede localizar Beagle –tanto por su nombre o contenido– no sólo son archivos en sí. Beagle también es capaz de acceder al contenido de nuestro correo electrónico, conversaciones instantáneas o historial de navegación. El resultado se muestra de forma resumida según el tipo de información, tal como se puede comprobar en la figura 35.

Figura 35



Un clic simple sobre cualquier de los elementos nos muestra más información sobre éste (figura 36).

Figura 36



Y un doble clic ejecuta la aplicación correspondiente para visualizar el elemento seleccionado. Todo ello con una velocidad sorprendente que permite que la búsqueda se vaya actualizando a medida que tecleamos las palabras clave.

El desarrollo de la aplicación Beagle ha supuesto un avance en la comunicación de Mono con el sistema operativo. Entre otros, la comunicación del código C# con el sistema de monitorización de cambios en el disco del sistema operativo. De esta manera, el motor de indexación subyacente de Beagle está al corriente de los cambios que se producen para actualizar el índice que ha creado inicialmente. La comunicación entre los distintos componentes se realiza mediante el nuevo estándar Dbus, propuesto por la organización FreeDesktop, que ahora también será adoptado por el escritorio KDE.

Bibliografía

Dumbill, E.; Bornstein Niel, M. (2004). *Mono: A Developer's Notebook*. (1.ª ed., julio, 21). Ed. O'Reilly Media.
ISBN: 0596007922

Mamone, M. (2005). *Practical Mono (Expert's Voice in Open Source)* (dic., 8). Ed. Apress.
ISBN: 1590595483

H. Schoenig, H.; Geschwinde, E.; Schonig, H. (2003). *Mono Kick Start* (1.ª ed., sept., 15). Ed. Sams.
ISBN: 0672325799

Web del proyecto Mono: http://www.mono-project.com

Web sobre Mono en español: http://monohispano.org

GNU Free Documentation License

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.

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.

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.