1. Introducción:

Cuando comenzamos a escribir algo de código en alguno de nuestros proyectos anteriores, encontramos algunos problemas a la hora de ponernos de acuerdo para nombrar nuestros ficheros y crear la jerarquía de directorios. Esto no representa un problema si se trata de un proyecto pequeño; pero si se trata de un proyecto con varios componentes, sería recomendable saber qué es lo que hace cada componente y saber que partes va a tener sin tener que perder ese tiempo en mirar el código fuente, utilizando nombres descriptivos.

Ahora, tenemos entre manos quizás el primer proyecto grande que sabemos que puede ser reutilizado en algún otro proyecto (common vfs). Y es bastante importante que tengamos estas cosas bien claras.

Como herramientas, en el subversion de mspace tenemos ya creado un build para NAnt, en scripts/nant.xml. Ese build es una implementación de referencia para la estructura que voy a detallar ahora.

2. Descripción:

Ahora que ya conocemos el problema que hemos tenido; mi propuesta es la siguiente. Es una propuesta que ya he utilizado alguna vez con resultados bastante buenos, a mí personalmente me gusta. También expondré las modificaciones que serían factibles de realizar.

2.1. Proyecto:

Utilizaremos proyecto como el programa que vamos a realizar, ese proyecto puede tener 1 o más componentes. Y cada proyecto debe tener un nombre lo suficientemente descriptivo de lo que será la aplicación. Sin tener miedo a poner nombres largos.

2.2. Comunes a proyecto:

Así pues, tendremos el directorio raiz, que será el user.dir. Ese directorio contendrá otros directorios, concretamente: lib, src, resources y luego además existirá un bin y un doc que se irán creando conforme utilizemos nant para compilar nuestro proyecto completo.

2.2.1. lib

Lib contendrá otras librerías externas de las que dependa nuestro proyecto, si por ejemplo vamos a utilizar la libreria log4net, deberemos copiar el ensamblado a lib. Otra opción que se barajará es si esa librería está instalada en el gac, como un strong assembly, pero eso ya es otro tema. En principio en esta IR no se da soporte de integración con el gac. Si ademas en lib vamos a utilizar otros mspace components, deberán ser copiados a lib/components. Con su ensamblado y además otro fichero que describa el interface del component (falta mucho por discutir sobre como sacar esos interfaces).

2.2.2. resources

Aquí vendrán todo aquello que sean recursos para la aplicación, todavía el build de nant no soporta la inclusión de resources en el ensamblado, pero en la próxima versión si que lo hará. Así pues podemos tomar como recursos, todo aquello que sean iconos, mensajes de i8n, scripts para ejecución de programas externos, logs, ficheros de configuración ...

2.2.3. src

El corazón del proyecto, aqui se guardará el código fuente de los componentes de la aplicación. Más tarde pasaré a hablar sobre el esbozo de modelo de componentes que he pensado y propongo utilizar en los componentes mspace, para alguien que conozca la framework guasaj, le resultará bastante similar.

2.2.4. bin

Aqui se guardaran los binarios de la aplicación, así como los pares de claves para firmar un ensamblado.

2.2.5. doc

Parece sencillo pensar que es lo que habrá aqui.

2.3. src

De nuevo volvemos a src, aqui se explicará como hay que organizar nuestro código fuente. Para no utilizar un sistema como el que utiliza java, es decir, que para cada paquete lleva implicito un directorio, dado que en .NET eso no es una restricción y además podemos hacer más accesible nuestro código fuente.

2.3.1. appname

Lo primero que encontraremos será un directorio con el nombre de la aplicación. Ese directorio contendrá *solamente* el lanzador de la aplicación, que sería conveniente llamarlo Launcher.cs y la clase debería ser Launcher. Así a secas, y en su main debería hacer una llamada al InitApp del maincomponent.

2.3.2. appname.Components

Este directorio contendrá el código fuente de los componentes que hayamos escrito a la hora de escribir la aplicación. Solo contendrá el nombre del componente que está escrito. Si por ejemplo quiero un componente que sea un conector TCPIP, y lo escribo ahi, deberia llamarse tcpipconnector (o algo así similar).

2.4. Jerarquia en un componente:

Dentro de cada componente y adaptandonos ya un poco al modelo de componentes, tenemos los siguiente subdirectorios: Bo, Dao, Exception, Factory, Form, Interfaces, Resources, Vo

2.4.1. Bo

Aquí se implementará toda la lógica de negocio de nuestro componente, no tiene porque contener sólo una sola clase. Pero deberá contener un punto de entrada al modelo que contendrá los casos de uso de nuestra aplicación, y esos casos de uso serán los que deben ser implementados de nuestro interface. Como se implementen esos casos de uso, es tarea del programador del componente. Puede utilizar más clases que estén dentro de Bo (la idea es tener todo el business object (Bo)) en este namespace.

2.4.2. Dao

Aquí estará todo aquello que sea un Data Access Object, es decir, objetos que sean para acceder a datos; esto es conexión con una base de datos, clases para leer xml. *No se debe acceder a Dao desde el Form!!!*

2.4.3. Exception

Aquí se meteran las excepciones que definamos nosotros para nuestro Bo y guasaj implementa también un gestor de esas excepciones con el método processException (Exception exception). El gestor de excepciones es útil para tener centralizado todo el código de nuestras excepciones y saber dónde se gestionaran los errores.

2.4.4. Factory

Aqui estarán las factorias que utilizaremos para crear los objetos, existen 2 filosofias que seguir acá (las que yo propongo). De entrada, como a mi me han dicho y explicado:

> Todo es un singleton hasta que se demuestre lo contrario.

Es preferible hacer todo un singleton y si necesitamos varias instancias ya no deberá ser un singleton (evidentemente). Y la otra es que cuanto menos llamemos a los constructores dentro del BO mejor, que mejor. Es preferible utilizar factorías para construir los objetos, dado que estos pueden ser extremadamente complejos y que a veces podemos necesitar cambiar un objeto en tiempo de ejecución. (Esto se aplica bastante bien a la hora de crear y utilizar un Strategy).

2.4.5. Form

Aqui se ubicará todo aquello que sea parte de la vista. No importa la framework que vayamos a utilizar, nos dará lo mismo utilizar GTK o Windows.Forms o wxWidgets (no han sido consideradas las WebApps). Tiene que ser transparente y el modelo de componentes para ello obliga a implementar un ViewDispatcher para hacer esto de forma transparente.

2.4.6. Interfaces

Aqui se ubicaran los interfaces que se deban implementar en las clases del Bo. Ya son de sobras conocidas las ventajas que aportan los interfaces a nuestras jerarquías de clases.

2.4.7. Resources

Los recursos *propios* de cada componente, es como el resources de más arriba, pero con un ámbito más reducido a cada componente. Probablemente también se empaqueten en otro ensamblado los resources.

2.4.8. VO

El value object es uno de los patrones más sencillos y poderosos que he conocido. Para los que hayan programado en Java a este nivel, les resultará muy similar a un Bean, pero bastante más sencillo. Simplemente estas clases son valores, es decir, estas clases no contienen acciones ni método, sólo propiedades. Un ejemplo claro puede ser la necesidad de crear una configuración y llamar a los métodos del Bo con una configuración especifica según como haya sido configurado. Asi pues utilizamos una clase ConfigurationVO, que contendrá por ejemplo un valor string Nombre getter; setter; y según sea ese valor se comportará de uno u otro modo la aplicación. Para rellenar los valores de aqui, es útil crearse un Dao que especificamente rellene el Vo. Por ejemplo si tenemos que utilizar una serialización XML de esos valores, es fácil hacer que se serialice o que acceda a los campos del xml el dao y no el Vo.

A. Sobre el modelo de componentes:

Aqui voy a dar unas pequeñas pinceladas sobre el modelo de componentes que vamos a utilizar para evitar el desacople. Está muy basado en Guasaj framework para Java, realmente yo cuando programo con Java ya sólo lo hago con Guasaj; por las ventajas que tiene, pero por supuesto también tiene carencias que he visto y que me gustaría solucionar.

La primera pregunta es: .NET ya tiene su propio modelo de componentes, ¿porqué no lo utilizas? El modelo de componentes de .NET es un buen modelo, pero quizás sea demasiado complejo para poder llevar a cabo estas aplicaciones, no sólo es un modelo de componentes, sino que también es un contenedor y el tema de los contextos no creo que sea necesario para realizar estas aplicaciones. También decir, que exactamente lo que se va a utilizar no es un modelo de componentes completo, sería algo así mas parecido a una framework para el modelado de componentes y la organizacion del código, ayudando al desacoplo entre componentes y manteniendo las demás características de una programación orientada a componentes. Compatibilidad binaria, gozando de independencia de lenguajes, un mspace component debería ser accesible desde cualquier lenguaje .NET, y por supuesto cada instancia del componente es única, cada componente debe ser un singleton (para variar).

No soy un experto en guasaj, pero he podido ver un poquito las ideas que esa framework conlleva, cada componente llevará unos casos de uso, y la clase que implementa esos casos de uso debe heredar de DefaultComponentHandler (en guasaj). Un componente se llama desde un configuration con el método getComponentByName (string cmpName); una vez conseguido el DefaultComponentHanlder del componente, utiliza el patrón executor para ejecutar los casos de uso, que por reflexión accederá a los interfaces y lo ejecutará, además de ejecutar una response a la vista donde haya sido definida en el interface. Es decir, el flujo de ejecución es el siguiente: Conseguimos el componente, y hacemos un

.execute del método con los parametros que sean necesarios ese caso de uso, en su interface xml lleva un response, que será ejecutado a continuación del caso de uso y ese response debe ser implementado en la clase que le digamos al interface (si no Exception) además cada componente lleva asociado un ExceptionManager que se encargará de localizar las excepciones y de tratarlas. Así pues conseguimos un desacople casi perfecto (por no decir perfecto) entre componentes y por supuesto un caso de uso puede responder a una vista, o a otra; incluso crear nuevas instancias de la vista con el ViewDispatcher y además asociado a ello control de concurrencia sobre los componentes.

Así pues, tendremos también un config indicando los componentes que existen y la localización de sus interfaces, logrando así tener nuestro ensamblado del componente por un lado, y por el otro la configuración.

Esto puede tener también un problema, que es que un mal uso del fichero xml puede llevar a errores, por este motivo el xml debe ser bastante sencillo para localizar los errores que nos pueda dar.

También tiene una restricción fuerte, pero espero que sea solucionada pronto con el vfs; es la posibilidad de invocar componentes que estén en otra máquina; pero esto es bastante más complejo y si queremos hacer esto no deberíamos utilizar esta framework, sino un modelo de componentes completo. Dado que nosotros no creo que vayamos a utilizar esto dado que las aplicaciones que vamos a realizar tienen un ámbito de aplicaciones de escritorio.

Otro problema que he visto es la notificacion de eventos, pero creo que esto es más por carencias del propio lenguaje Java frente a .NET y tener event como un tipo propio del lenguaje. Con este event podemos hacer estas notificaciones de una buena manera, pero *no se deben definir eventos en el interface!!* debe ser algo más interno al componente.

Finalmente el problema puede ser la redireccion a varias vistas lo cual el xml debe proveer un medio para decir a que vistas debe ser dirigida la response de un caso de uso. Lo cual, ampliando la definición del xml debe ser una tarea sencilla.

B. Requerimientos

La técnica que se va a utilizar conlleva un uso bastante grande de reflexión esto no es ningún problema.

El xml debe ser tratado de un modo rápido, pero dado que se debe permitir que los componentes no sean grandes, sino que sean algo parecido a piezas de un puzzle en nuestra aplicación, el poder utilizar DOM para parsear el xml no es ningún problema.

Además es preferible en todas las clases que estén fuera del ComponentHandler, hacer un get del componente para realizar las acciones; es decir, si en el form tengo un boton que lo clicko y debe realizar un caso de uso del componente, no llamar al Bo; sino que obtener el componente y utilizar el executor que lleva.