Guasaj. Un framework de programación basado en componentes

Benito Mateo, Urko
http://sparcki.blogspot.com/
itilys@yahoo.es

Blesa Jarque, Ángel
http://ablesa.wordpress.com/
angel.blesa@gmail.com

Lop lis, José Javier
irracional79@gmail.com

Abstract

En el ciclo de vida de un proyecto, independientemente del proceso de desarrollo que se utilice (RUP, FDD, XP, etc...) en un determinado momento se debe pasar del análisis de los requisitos al diseño e implementación del mismo.

Los patrones de diseño hacen colaborar un conjunto de clases e instancias de las mismas, para solucionar un problema determinado permitiendo obtener una solución con buenas características de acoplamiento, cohesión, granularidad, rendimiento, adaptabilidad, evolución, etc. , pero con los patrones de diseño sólo no basta, no dicen directamente que modelo superior crear para hacer colaborar N paquetes de casos de uso, y que permanezcan desacoplados, independientes, reutilizables, etc..

En la búsqueda de este modelo superior surge el componente de software como unidad independiente, cooperativa, desacoplable, etc.

Lo que pretendemos con este proyecto es proponer una metodología de organización de código en torno a componentes con una estructura basada en patrones de diseño bien conocidos(MVC, Observer, ViewDispatcher, Factory, Command, Facade, VO, Service Activator , Service Locator, ...) que nos lleve del análisis de los casos de uso (historias, requisitos funcionales, casos de aplicación, etc...) a la implementación de la solución en código de una manera metódica, repetible y con resultados reutilizables en diferentes proyectos..

Keywords: framework, patrón, componente, metodología.

1 Motivación. Origen de guasaj.

El origen desde el cual guasaj fue concebido fue crear un entorno de trabajo en el cual se defina un procedimiento para plasmar el análisis y diseño de la solución de un problema en un conjunto de componentes reutilizables.

Con ello se pretende aumentar la productividad a través de dos cuestiones:

- El establecimiento de un procedimiento claro de implementación de las funciones descubiertas en el análisis y el diseño.
- La reutilización automática del trabajo realizado en el paso anterior.

En primer lugar se pretende crear un esquema de clases sencillo para la implementación de componentes, con un ciclo de vida, en la llamada a sus servicios, lógico y que cumpla las premisas de los patrones de diseño sobradamente conocidos [1] (MVC, command, service locator, factory, etc...). A través del estudio de la filosofía de creación de aplicaciones basadas en componentes pasar a una implementación de la misma, sencilla, pero potente y flexible, creando una asociación rápida y semiautomática entre el modelo conceptual de una aplicación y su correspondiente implementación.

Con este proceso se pretende eliminar, dentro de lo posible, el problema de tener que "reinventar la rueda" en cada proceso de análisis y diseño de un determinado problema, muchas veces supeditado a la imaginación e inspiración en el diseño final del arquitecto/diseñador/programador.

Otro de los factores o puntos de especial interés radica en el hecho de establecer una definición, implementación, manejo, capacidades, y ciclo de vida

de los componentes, de tal manera que todo el equipo de desarrollo emplee el mismo idioma.

En definitiva, ir un paso más allá de los patrones de diseño en el establecimiento de un lenguaje común para la nomenclatura de las funciones, comportamiento y arquitectura de un sistema.

2 Problemas detectados en el desarrollo de programas en la actualidad

En la labor de desarrollo de proyectos software se presentan una serie de problemas con demasiada frecuencia, como son:

- Dificultad en el paso de análisis, al diseño y codificación. La identificación y agrupación de las funcionalidades del sistema en paquetes y su posterior paso a codificación no está suficientemente sistematizado.
- Poca o ninguna aplicación de patrones de diseño para resolución de problemas genéricos en los proyectos cotidianos, debido en parte a la dificultad para abstraer la problemática concreta de un gran proyecto, a las soluciones a un nivel más micro que aportan los patrones de diseño. Se hace complicado pasar de pequeñas implementaciones de soluciones en una serie de clases y objetos aplicando unos determinados patrones, a la implementación de una serie de paquetes o componentes que desarrollen una parte concreta, independiente, reutilizable, extensible y configurable de la aplicación de un tamaño considerablemente mayor.
- No existe una separación clara entre la vista de la aplicación y el modelo, sucumbiendo en la mayoría de las veces ante la dificultad de disociar las distintas porciones de código que participan en una parte de la solución, interface gráfico, eventos del GUI, validación de datos, acceso a datos persistentes , etc …
- Dificultad de programación para el interface en lugar de la implementación.
- Mala gestión de la creación – instanciacion de objetos tanto para ejecutar la lógica del sistema, como para representar el estado del mismo en un momento dado.
- No se define una separación clara entre al estructura de datos que almacenan el estado del sistema en cada momento de ejecución del mismo, y las diferentes operaciones que se pueden realizar sobre dichos datos.

No existe, en general, un procedimiento sistemático para codificar la funcionalidad que se requieren (casos de uso, historias, requisitos arquitectónicos, características funcionales y no funcionales, etc), de tal manera que mantengamos desde el principio un separación entre vista y modelo, una gestión clara de los datos del proceso, un manejo de excepciones efectivo, y una posibilidad de reutilización practica y rápida, desde el principio, de la codificación, de tal manera que la exportación de funcionalidades desde un proyecto a otro sea automática.

3 Del análisis a la programación. Pasos y trabas intermedias

En cualquiera de las metodologías o procesos de desarrollo de software, salvando sus peculiaridades, se sigue un proceso similar al que se describe a continuación para la consecución de la solución a un proyecto:

- Estudio de los requisitos.
- Análisis de la solución. Modelo conceptual. Descubrimiento de las entidades del problema. Agrupación en paquetes conceptuales.
- Diseño de la solución. Patrones. Establecimiento de la arquitectura. Diseño de pruebas. Cumplimiento de requisitos no funcionales, etc…
- Codificación. Llegado este momento en determinadas metodologías podemos estar en un estado más o menos avanzado en la identificación de clases y métodos a implementar, pero en general se dista bastante de tener una visión clara y lo más importante, acertada y con posibilidades de ser la definitiva. Esto deriva en una lógica de negocio más o menos dispersa en métodos de clases cuyas responsabilidades no están claramente definidas. Esto puede llevar a gravísimos problemas de acoplamiento entre partes funcionales del sistema, difíciles de independizar a posteriori, siendo muy difícil cuando el modulo en cuestión tiene ya un tamaño considerable y nos percatamos de las limitaciones impuestas, pensar en una

refactorización del diseño que no nos provoque más de un dolor de cabeza y desde luego una merma en nuestra productividad.

3.1 Solución. Aplicación de patrones

Los patrones nos aportan buenas prácticas de código, en la pequeña escala, para salvar de una manera óptima y muy probada circunstancias que se dan en la programación cotidiana.

El uso de patrones hace que las porciones de código resultantes sean robustas, flexibles, extensibles, adaptables, optimizadas y fiables, de tal manera que el conjunto de la aplicación también herede dichas cualidades. Esto es lo que se pretende pero en el camino para lograrlo se tiene una serie de imponderables, que hacen que las decisiones de diseño no sean triviales.

En el momento de diseñar la solución para una aplicación el arquitecto/diseñador debe tener una concepción de las entidades y la colaboración entre las mismas que le harán desempeñar la solución requerida, así como una capacidad de abstracción para descubrir entre estas entidades la problemática que encierran y que parte de ésta, está contemplada como patrón sobradamente conocido.

3.2 Problemática de los patrones en la codificación

El uso de patrones de diseño en la solución de una aplicación facilita el paso del diseño a la implementación por estar estos ya codificados en multitud de ocasiones. Aún así, hay un nivel de abstracción intermedio en las soluciones genéricas que aportan los patrones, y los problemas de modelado, diseño, empaquetado, y reutilización de código que se necesitan en los proyectos con una tamaño considerable, sobre todo en sistemas en los que se desarrollan una serie de aplicaciones *(suite)* para dar una serie de servicios integrales a una empresa.

La solución puede pasar por la elaboración de un *framework* que encapsule las buenas prácticas de los patrones sobradamente conocidos y haga que los programadores sean capaces de representar el modelo lógico de la solución del problema, expuesto en el análisis, con un buen diseño, una buena arquitectura y un código ampliamente reutilizable sin tener que implementar físicamente los patrones, sino que sea el framework el que *maquille* para el desarrollador final el uso de los mismos. Dicho de otra manera, el hecho de programar bajo este framework garantiza la herencia de la mayoría de los patrones de diseño más utilizados, sin prohibir en absoluto la implementación de alguno de los mismos para cubrir una determinada necesidad.

4 Condiciones deseadas en la solución

Un framework que pretendiese implementar una solución para la problemática anteriormente descrita deberá poseer una serie de características como:

- Guiarse por un patrón de comportamiento como el MVC para la separación de vista, modelo y controlador.
- El código resultante debe quedar en forma de componente reutilizables, y cumplir con el paradigma y filosofía de construcción de aplicaciones basadas en componentes tal y como se detalla más en profundidad en [2].
- El cliente de un componente estará desacoplado del mismo, ni siquiera a través de una interface. La resolución del componente y del método a ejecutar se realizará en tiempo de ejecución
- Los componentes que participan en una aplicación, la descripción de los mismos en cuanto a comportamiento y aspectos (acceso, relaciones, vista, excepciones, log) debe tener un soporte de configuración exterior al código (archivo xml de configuración).
- Los componentes deben permitir la ejecución secuencial de dos métodos de distintos componentes, o desde uno de estos a otro. El hecho de que un método de un componente sea llamado desde otro puede provocar en determinadas circunstancias que la respuesta de éste sea de un tipo o de otra
- Un componente tiene una lógica de negocio representado por un BO (Bussiness Object) con los métodos operativos, y además un componente trabaja con un conjunto de datos en forma de VO (Value Object).
- El estado en memoria de un componente en un determinado momento de ejecución (por ejemplo, un servidor de Chat, con una serie habitaciones y

unos usuarios en las mismas, en un determinado momento) se debe reflejar en este BO. Pero no una instancia del BO por cada elemento a reflejar. Por ejemplo, el componente "Habitación" en una aplicación de chat, estaría formado por un BO con las operaciones y una lista de posibles habitaciones.

- De esta manera un componente será gestor y almacén de los datos que puede manejar. Los componentes son elementos de peso de tal manera que en el sistema no tendremos múltiples instancias de un componente[1], sino que será el componente el que maneje los datos necesarios y trabaje con las operaciones para construir y manejar el estado del sistema en un momento dado.
- Cuando en una vista de un componente se lleve a cabo un cambio, dicho cambio deberá ser reflejado de alguna manera en el estado de la lógica del componente, en su BO, es decir, un cambio en una de las vistas se ve reflejado en su homónimo del modelo. Por ejemplo, cuando en un componente gestor de barra de herramientas seleccionamos un elemento y desde otro componente, queremos acceder al elemento seleccionado. No podemos desde una vista de un componente acceder directamente a la vista de otro, debemos pasar por el modelo y solicitar algún atributo que nos de la información que requerimos.

5 Características de los componentes guasaj

Un *componente guasaj* es un conjunto de clases diseñadas para llevar a cabo una serie de funciones relacionadas con un problema del modelo de negocio del sistema o desarrollar una función de carácter más amplio dentro de la arquitectura, estando este acompañado de un fichero de descripción de las características y posibilidades de ejecución de dicho componente. Ejemplo, *roles de usuario, pre y post ejecuciones, controlador de excepciones, métodos virtuales…*

Un componente guasaj es autocontenido en su ciclo de vida, llevando a cabo una separación entre lógica de negocio y datos, así como vista y modelo. En un principio la plataforma objetivo de guasaj es la J2SE aunque su filosofía es extrapolable tanto a J2EE como J2ME. Todo componente aporta un mecanismo de llamada a sus servicios, un método de gestión y acceso a sus datos, así como un control de sus vistas, manejo de excepciones y control de acceso.

Dentro de los tipos de componentes que podemos desarrollar se pueden encontrar 2 tipos principales con sus peculiaridades y variantes:

Componente entidad: Son componentes que intentan representar el modelo lógico de negocio propio de un proyecto asumiendo una serie de características de la solución final. Estos componentes desarrollan una función, tienen una responsabilidad y unos mecanismos o reglas de interacción con otros componentes de entidad o de servicio. Un ejemplo de componente representativo de una entidad del modelo de negocio podría ser los componentes *cliente, factura, proveedor, empresa, albarán*, etc... por ejemplo, en una aplicación de gestión. En un sistema de tratamiento de señales provenientes de un sistema de instrumentación electrónica, estos pudieran ser, *equipo, señal, punto de medida,* etc. En general los candidatos a este tipo de componentes pueden ser los provenientes del análisis conceptual del dominio del problema.

Componentes genéricos de servicio: Componentes que permanecen completamente independientes de las peculiaridades de un determinado proyecto siendo totalmente reutilizables en otro ámbito de ejecución e incluso plataforma, si están diseñados para ello. Por ejemplo, un componente para comunicaciones TCP (cliente y servidor), un componente para selección y gestión del idioma para una aplicación, un componente manejador del estilo y fuentes de una aplicación, un componente calendario, un componente calculadora, etc…

De esta manera en la elaboración de una solución completa para un problema, partiremos del modelo tradicional de resolución de problemas software, de la siguiente manera:

- Se plantea una serie de requisitos o funcionalidades a desarrollar.
- Se lleva a cabo un análisis funcional y se determina un modelo conceptual en el que se descubren entidades, relaciones entre estas y atributos de las mismas.

- Se realiza una agrupación en paquetes guiándose por una determinada premisa pudiendo esta ser, agrupación por función, agrupación por complejidad, agrupación por arquitectura, agrupación por departamento lógico del cliente, etc, etc.. Suele ser en este momento donde empiezan a surgir los problemas separación de responsabilidades entre las entidades del sistema. Aquí evidentemente no hay magia que valga, la decisión final de agrupación, separación de responsabilidades de cara a un diseño u otro es propia del arquitecto /diseñador, siendo muchas veces equivalentes soluciones con topologías de entidades diferentes.

Una cuestión que puede y debería ayudar sobremanera en este paso del proceso de desarrollo, sería el hecho de saber exactamente el procedimiento de codificación que seguiremos para llegar a implementar esta entidad, e igualmente los servicios a los que tendremos acceso sólo por el hecho de implementar la entidad de acorde a este procedimiento / framework de programación.

La idea radica en crear una equivalencia entre cada uno de los términos utilizados en el análisis funcional de la aplicación y la codificación de los mismos. Saber medir que implicaciones tendrá el hecho de utilizar palabras como paquete, clase, entidad, atributo, relación, caso de uso, servicio, etc.… en la codificación del sistema.

6 Introduciéndonos en guasaj

De esta manera y según lo expuesto fundamentos de los cuales parte el desarrollo de guasaj son:

- Separación en modelo, vista y controlador.
- Independencia de componentes.
- Variación del comportamiento de un componente según una definición exterior, sin modificar el código.
- Estandarizar la nomenclatura y tipo de clases al crear un componente reutilizable.
- Crear un elemento amigable tanto para el analista como para el programador del sistema, pasando por el arquitecto y el diseñador.

6.1 Localización de componentes por nombre.

Para tener acceso a los servicios de un componente debemos primero localizar una instancia de dicho componente. Para ello, pasaremos un nombre único con el que denominaremos unívocamente a un componente dentro de la aplicación y estará referido en el fichero de descripción guasaj y del componente.

Una vez localizado un componente y apuntado por una interfaz común para todos podremos llamar a los servicios que implementa.

6.2 Ejecución de métodos de los componentes dinámicamente.

Resolución de los métodos a ejecutar según la firma del método, nombre y paso de parámetros, pudiendo derivar en diferentes comportamientos en la ejecución y la respuesta según los parámetros pasados.

6.3 Posibilidades de redirección a la vista.

Cuando se llama a un método de un componente, se quiere llevar a cabo una funcionalidad y generalmente obtener un resultado de la misma, acompañada de una visualización de los resultados. Otras veces, se quiere ejecutar la lógica de modelo pero no se quiere visualizar el resultado de la misma a nivel de interface gráfico, sino que sólo se quiere manejar los datos del resultado de la operación. De esta manera se puede elegir en el momento de la llamada a un método de un componente si queremos ejecutar la parte de vista vinculada a ese método o no. Igualmente, podemos indicar en el momento de la llamada qué método y de qué componente queremos que se haga cargo de manejar la respuesta en forma de vista de ese método. Y aún hay más, podemos decirle en que instancia de dicha vista queremos que se ejecute el método de respuesta, en el caso de que manejemos diversas instancias de vista de la misma clase.

Con todo lo anteriormente expuesto, tenemos tres funciones relacionadas con la vista:

- Primero, indicar si queremos manejar vista o no.
- Segundo, indicar que método de que componente queremos que maneje la respuesta en forma de vista.

- Tercero y último, si para ese componente queremos instanciar una nueva vista o manejar una que pasamos en la llamada al método. Tres en uno.

Así, teniendo la ventaja de una separación entre componentes, podemos acceder a través de las funciones del contenedor (éste las resuelve en tiempo de ejecución, no de desarrollo), a otros componentes para pasarles (notificarles) resultados de operaciones de otros componentes. Esto corresponde con la teoría de inversión de control [3], por la cual es el contenedor el que llama a nuestras clases para realizar ciertas funcionalidades en un momento dado.

6.4 Posibilidad de llamar a métodos no bloqueantes.

Sin tener que esperar a que termine la ejecución del método llamado. Esta utilidad resulta interesante cuando queremos tener un mecanismo automático de ejecución en segundo plano de una operación y recibir la respuesta de esta cuando finalice la misma, sin que tengamos por ello bloqueado el flujo principal del programa, esperando retornar de la llamada efectuada. Un ejemplo concreto de esta funcionalidad seria la localización remota de un servicio o componente que puede llevar unos segundos. Es posible que no podamos seguir la ejecución de algo concreto sin este componente o servicio pero eso no impide que podamos ejecutar otras opciones del programa y ser avisados cuando se haya localizado el componente o cuando haya vencido el plazo de *timeout*.

6.5 Concentración del manejo de excepciones.

A través de un *HandlerException* por componente. Mecanismo por el cual las excepciones producidas son dirigidas a un método de una clase que se ocupa de su gestión y tratamiento. También existe la posibilidad de definir si el trato de excepciones se procesa en el componente original o si se redirecciona a otro manejador en el proyecto destino (ya que dependiendo de cómo y dónde se ejecute un componente puede que no queramos ni deseemos su tratamiento original o incluso necesitemos capturar las excepciones producidas por un componente y tratarlas a nuestro modo).

6.6 Control de acceso a los métodos y componentes por rol de usuario.

A través del fichero de descripción del componente se puede indicar que roles de usuario tendrán acceso a cada método. Lo único que se debe realizar es activar en el contenedor guasaj el rol de usuario activo. En tiempo de ejecución comparará si el rol activo coincide con alguno de los que tiene permiso de ejecución. El proceso común será aprovechar el momento en el que pidamos validación al usuario para recoger de este el rol de usuario y activarlo en guasaj. Para este usuario cuando ejecute el resto de operaciones su nivel de acceso vendrá determinado por dicho rol.

6.7 Métodos virtuales.

Son métodos que no está implementados en el componente original, sino que estarán redireccionados a otro método de otro componente que será el que realmente se ejecute, recibiendo los parámetros originales. Esta función es necesaria para permitir que las llamadas a métodos realizadas desde el "interior" del componente puedan ser capturadas y manejadas por otro componente. Por ejemplo, tenemos un componente que gestiona conexiones TCP, atiende a los clientes conectados, y recibe peticiones de los mismos. Este componente es reutilizable en multitud de proyectos, pero no sabemos en tiempo de implementación del componente que método de que clase tenemos que llamar cuando recibamos una trama, ya que, evidentemente, dependerá de lo que queramos hacer con la misma en el proyecto que nos ocupe. Para ello podemos definir un método virtual en este componente, *receiveData,* que será invocado por cada hilo gestor de un cliente conectado, para indicar que nos ha llegado una trama de datos. En el fichero de descripción de componente indicaremos que éste método es virtual y qué método de qué componente queremos que se ejecute cada vez que se llame a dicho método virtual.

6.8 Preejecución, postejecución e intercepción de métodos.

En la descripción de un método, dentro del archivo de configuración del componente, podemos especificarle un método que se ejecutará antes que este. Este último recibirá los parámetros del método original para su ejecución. En el caso de queramos una postejecución,

podemos definir un método que se ejecutará con posterioridad a la ejecución del método original. En este caso este método recibirá tanto los parámetros del método original como la respuesta producida por este. Y por último, en el caso de la intercepción, el método interceptor recibe los parámetros del original, puede modificar los mismos, y llamará al método original con éstos. Un ejemplo de preejecución, puede ser incluir un sistema de logs a un determinado método. En el caso de la postejecución, un posible ejemplo sería la elaboración de un informe al término de la ejecución de un método, necesitando para ello, los parámetros de entrada, respuesta del método original y los posibles cambios que haya podido originar dicha llamada, como escritura en base de datos, cambios de estado, alteración de ficheros o configuraciones, etc…

Y por último, un ejemplo de intercepción sería el hecho de dotar de un sistema de cifrado a un mensaje pasado entre componentes. Podríamos capturar los datos de la llamada original, cifrarlos o descifrarlos y continuar con la llamada en el componente destino que se ejecutaría ya con los datos cifrados o descifrados.

Los métodos, pre, post e interceptor son ejemplos claros de componentes que dotan una funcionalidad añadida no contemplada anteriormente en el sistema.

7 Otras consideraciones en los componentes guasaj

Los componentes guasaj están concebidos desde un principio con una vocación de reutilización de los mismos en diferentes ámbitos y soluciones informáticas, sin por ello acarrear un malus de productividad para el usuario final de los mismos como desarrollador de aplicaciones guasaj. Al contrario al pensar directamente en la reutilización del trabajo realizado se tienen en cuenta desde las fases más tempranas de codificación unas características de arquitectura y diseño muy positivas. La clave de todo este proceso resulta en no crear una abstracción vacía de elementos software de difícil implementación sino todo lo contrario, dotar de un framework que nos haga una gran parte del trabajo de nomenclatura de nuestras clases, separación de responsabilidades, posibilidades de relación entre componentes y acceso a los servicios de los mismos desde cualquier parte de la aplicación.

8 Conclusiones

En la ejecución de un proyecto software, el paso del análisis del problema al diseño y codificación presenta una serie de dificultades para asignar las diferentes responsabilidades entre las unidades de código que vamos creando. Dentro de las unidades de código queremos mantener una separación entre la lógica de negocio y la vista. Gran parte del código que generamos para la solución de un determinado problema deberíamos poder reutilizarlo en parte o por completo, con alguna pequeña variación de configuración en otros proyectos, dentro del mismo ámbito o no. El componente guasaj nos permite crear estructuras de código que desarrollen un conjunto de funciones reutilizables en otros proyectos. Los patrones de diseño por si mismos no establecen un lenguaje suficiente para la nomenclatura de la funcionalidad, comportamiento y arquitectura de un sistema. Debemos ir hacia una entidad de mayor calibre que cubra esta necesidad.

Resulta fundamental definir un conjunto, framework - procedimiento, que automatice el proceso de codificación del sistema, sin depender en exceso de la inspiración particular, para llegar a lograr un diseño arquitectónico correcto, repetible y reusable, todo ello en aras de una mayor productividad y calidad software.

Agradecimientos

Queremos agradecer a la EUPLA la posibilidad de contribuir a este congreso con el desarrollo de este trabajo. Igualmente a ARCO ELECTRÓNICA S.A, las facilidades para la realización del mismo, y la asistencia a este congreso. También un agradecimiento especial para *Sergio Gil García* por contribuir en los inicios de esta idea.

Y en especial a la comunidad javaHispano por permitirnos participar en este congreso.

Referencias

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns. Element of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[2] Claudia Patricia García Zamora y Samuel Garrido. Programación basada en componentes. CINVESTAV México D.F 4 - Noviembre -2003.

[3] Mike Spille, Inversion of control *http://www.pyrasun.com/mike/mt/archives/2004/11/06/15.46.14/index.html*