Universidad de las Ciencias Informáticas
Facultad 3

Tema: Propuesta de una guía metodológica de buenas prácticas orientada a la calidad del proceso de diseño de software.

Trabajo de Diploma para optar por el título de Ingeniero Informático

Autor: Jorge Luis Valdés González

Tutor: Ing. René Lazo Ochoa

Junio 2008

AGRADECIMIENTOS

A mi madre en primer lugar por ser el centro de mi vida.

A mi familia toda por el apoyo sistemático durante toda mi carrera.

A mi padre biológico y amigo de toda la vida Jorge Orlando, por su aporte dada su experiencia en el desarrollo de software, por la ayuda en el soporte bibliográfico y por estar siempre ahí.

A la esposa de mi padre, amiga para toda la vida, Daniela, por su ayuda, por ser una persona tan especial y por muchas veces desde el anonimato, hacer posible que este trabajo se realizara.

A mi tutor Ing. René Lazo Ochoa por su apoyo, guía, persistencia en la calidad del trabajo, por su aporte científico-técnico, por estar siempre dispuesto, por su experiencia y por ser un amigo.

A mis compañeros de misión en Venezuela Joisel, Raymond y Lianet por su ayuda.

Al Doctor en Informática por la Universidad de Salamanca Francisco José García Peñalvo, Profesor Titular de Universidad en el Departamento de Informática y Automática de esta Universidad, Director del Grupo de Investigación en Interacción y *eLearning* y Vicedecano de Innovación Tecnológica en la Facultad de Ciencias, por su ayuda en los temas de reutilización de software.

Al Máster en Ciencias José Rodríguez de la Universidad Autónoma de México, por su colaboración en los temas relacionados con la problematización del proceso de trabajo en los programadores de Software con una propuesta desde la perspectiva de la Sociología del Trabajo.

Al Ing. Eduardo Díaz, especialista de PDVSA, por hacer posible una entrevista con la empresa desarrolladora de software venezolana E4GS, para abordar temas relacionados con las factorías de software y el proceso de diseño.

Al Ing. Ramón A. Espinosa, especialista en desarrollo de software y miembro del departamento AIT-PDVAL de PDVSA, por colaborar con el proceso de validación del modelo.

Al Ing. José Yván Bohorquez, Director de la empresa venezolana desarrolladora de software E4GS, por ceder parte de su tiempo a brindar las experiencias de mercado de la empresa y cómo desarrollaban los diferentes procesos de construcción de sistemas para la Internet dentro de una *factoría* que tenían definida dentro de su modelo de producción.

DEDICATORIA

A mi madre.

"(...) Después de escalar una montaña muy alta, descubrimos que hay muchas otras montañas por escalar (...)"

Nelson Mandela

RESUMEN

El presente es un trabajo consistente en la realización de una investigación sobre el proceso de diseño dentro del propio proceso de desarrollo de software. El mismo propone una guía metodológica de buenas prácticas orientada al mejoramiento de los atributos de calidad del diseño que además incluye importantes análisis relacionados con el correcto empleo de patrones, la evaluación de métricas orientadas al diseño como producto y como proceso, la estructura organizacional de un *equipo* de diseño, propuesta de roles de trabajo, artefactos generados por el proceso y una conceptualización enfocada en *perspectivas* o *dimensiones de diseño* desde un punto de vista del marco histórico, marco tecnológico, interrelación de elementos de software, rendimiento y seguridad e interfaces de comunicación.

ÍNDICE

Capítulo 2:

Bibliografía:

Anexos:

INTRODUCCIÓN

La actividad de diseñar en la vida del hombre ocupa una posición importante por ser un proceso previo de configuración mental. Mediante éste, se proyectan ideas reflejadas en elementos gráficos sobre diferentes tipos de soportes (digital, analógico o virtual) y se orienta a la solución de problemas de la cotidianidad humana.

Diseñar, en el contexto de la industria del software, puede verse en diferentes modalidades a través de diversos métodos: diseño arquitectónico, diseño de la interfaz de usuario, diseño a nivel de componente y diseño orientado a objetos. Sin embargo, podrían citarse muchos más métodos o categorías de diseño dentro del propio proceso de desarrollo de software.

El diseño arquitectónico, es el proceso mediante el cual se representa la estructura de los datos y los componentes del programa que se requieren para construir un sistema basado en computadora. Define la estructura y las propiedades de los componentes del sistema además de encargarse de establecer las relaciones existentes entre dichos componentes.

Por otro lado, el diseño de la interfaz de usuario, es una categoría de diseño encargada de crear el medio de comunicación visual entre el hombre y la máquina para su interacción con el sistema. Mediante un conjunto de principios de diseño, se identifican los objetos y las acciones de la interfaz, creando entonces un formato de pantalla que deviene en prototipo de la interfaz de usuario final.

En el caso del diseño a nivel de componentes, conocido además como *diseño procedimental*, se encarga de llevar el diseño de datos, de arquitectura y de interfaz a un software operacional, bajando un poco más el nivel de abstracción y centrándose esencialmente en las relaciones entre los elementos de software y la manera en que éstos interactúan (mensajes que se envían).

El diseño orientado a objetos se ocupa de transformar el modelo de análisis en un modelo de diseño basándose en los principios del paradigma orientado a objetos y que sirve como anteproyecto para la solución del software.

El diseño del software se encuentra en el núcleo técnico de la ingeniería del software y se aplica independientemente del modelo de diseño de software que se utilice. Una vez que se analizan y especifican los requisitos del software, el diseño es la primera de las tres actividades técnicas (diseño, generación de código y pruebas) que se requieren para construir y verificar el software. Cada actividad transforma la información de manera que dé lugar por último a un software de computadora validado (PRESSMAN, 2001).

Existen disímiles problemas en el proceso de diseñar un software de computadora. Algunos de los problemas más frecuentes son la **poca flexibilidad y extensibilidad** de los diseños de clases, lo que imposibilita la capacidad adaptativa y de reutilización de los módulos o unidades de codificación.

Otro problema que se observa es el **alto acoplamiento** entre los módulos que encarece el mantenimiento debido a la marcada dependencia entre ellos, y dificultan las posibilidades de integración entre subsistemas codificados y otros sistemas informáticos que podrían ser afines con el que se está desarrollando.

Así también la **baja cohesión** del diseño de subsistemas o clases de los mismos, provoca la reprogramación de funcionalidades, influyendo de forma negativa en la productividad y en la calidad, encareciendo el mantenimiento, la adaptación a cambios y la reutilización de las unidades de software en desarrollo. Además de esto, tiene relación con el incorrecto uso de la *abstracción* ya que se le asignan responsabilidades a objetos, las cuales podrían estar *encapsuladas* en una clase en específico.

Estos son algunos de los problemas más frecuentes, cuya principal causa está determinada por el poco o mal uso de patrones de diseño, que en muchos casos no

son ni siquiera tenidos en cuenta para estructurar el diseño de un software, tratando de "inventar" aspectos previamente definidos en patrones, violando así uno de los principios básicos del proceso de diseño.

El Diseño de Software como proceso, constituye un elemento fundamental para la realización "física" de un sistema en cuestión, transformando los requisitos funcionales del cliente en relaciones entre elementos de software. Si bien es cierto que existen otros flujos de trabajo que influyen en el proceso de obtención de un software, éste en gran medida determina la materialización de una solución de software cuyas funcionalidades y características se ajusten a las necesidades de los usuarios.

El diseño, en esencia, es un proceso caracterizado por estar dividido en diferentes niveles de abstracción de acuerdo a la categoría de diseño que se especifique, cuya finalidad es traducir los requerimientos de los clientes en una representación modular de lo que sería una implementación concreta de software con un alto grado de dependencia con la tecnología de desarrollo que se seleccione. Es la primera etapa técnica que se encarga de transformar un *lenguaje formal-natural* (lenguaje humano) en un lenguaje técnico de modelado y basado en una tecnología específica, que puede ser entendido por los encargados de implementar el producto de software.

En el momento en que se comienza a definir un diseño, es importante prestar la mayor atención a los atributos de calidad por los cuales será medida la calidad del mismo. Se puede hacer apoyándose en las métricas para el diseño de software además de otros mecanismos que validan la obtención de un resultado con calidad. Esto constituye un elemento esencial y al mismo tiempo, un problema real en el desarrollo de la dinámica del proceso de diseño en los proyectos de construcción de software.

Partiendo de la idea de que los errores son malas modelaciones de los requisitos del usuario y que estos no son detectados hasta el momento en que se comienzan a realizar las pruebas de software, es importante tener tres consideraciones respecto a

este tema: primero, que todos los requisitos deberán ser tenidos en cuenta para un correcto modelado del dominio; en segundo lugar, que los patrones de diseño sean incluidos en el proceso y además que la selección de éstos se realice partiendo de los más adecuados según las características del diseño y por último, que el diseño cumpla con sus principios básicos para que la navegabilidad por todo el proceso en sí mismo sea viable. A todo esto se le suma la evaluación sistemática de los atributos de calidad del diseño –desde un principio, como se mencionaba anteriormente- a través de las diferentes normas de calidad y dividido en dos enfoques: a partir del propio diseño como proceso, y a partir del diseño como producto.

Existen un sinnúmero de metodologías de desarrollo de software que hacen posible seguir una serie de pasos para llevar a cabo el proceso de diseño, sin embargo, ninguna metodología es ni única ni universal, ya que siempre hay que adaptarlas al contexto del proyecto según recursos humanos y técnicos, tiempo de desarrollo, tipo de sistema, tamaño del proyecto, etc. Según Pressman, "*Las metodologías de diseño del software carecen de la profundidad, flexibilidad y naturaleza cuantitativa que se asocian normalmente a las disciplinas de diseño de ingeniería más clásicas*" (PRESSMAN, 2001).

De los dos tipos de metodologías que existen, clasificadas en dos grandes grupos: las ágiles o livianas y las pesadas o tradicionales, existe una marcada diferencia, entre otras muchas cosas, reflejada en el modo de concebir el diseño. Por una parte, las metodologías ágiles confían en la creatividad del equipo de desarrollo y por ende, en la creatividad de los diseñadores, recargando en estos la mayor responsabilidad sobre la calidad del diseño. Por otro lado, las metodologías pesadas norman con mayor énfasis el proceso de diseño, definiendo un conjunto de artefactos que tributan a la calidad del mismo. Sin embargo, ninguna le da un enfoque en perspectivas para facilitar el entendimiento del proceso, como se propone en el presente trabajo: *Perspectiva de Marco Tecnológico, Perspectiva de Interrelación de elementos de software, Perspectiva de Rendimiento y Seguridad y Perspectiva de Interfaz*. Sin embargo, la idea no es ver al diseño por partes sino variar el objeto de análisis de

acuerdo a la perspectiva de diseño en la cual se centre dicho análisis. Esto permite un mejor entendimiento del proceso y por tanto una mejor identificación de qué es lo que se tiene que hacer en cada momento y con la ayuda de una guía o método, cómo se debe hacer, además de cómo y cuándo evaluar.

Cada uno de los proyectos de software que se desarrollan en el mundo, siguen alguna metodología en particular. Lo real es que los proyectos de desarrollo de software en Facultad 3 de la Universidad de las Ciencias Informáticas, por ejemplo, no están exentos de esto. En cada uno de ellos se aplica alguna metodología que dicta *cuándo* hacer *qué* y por *quién*. Sin embargo, ninguno cuenta con una guía metodológica de buenas prácticas de diseño que permita elevar la calidad de los productos de software obtenidos, reflejada de manera positiva en el nivel de abstracción, la reutilización, flexibilidad, facilidad de mantenimiento del diseño, adaptabilidad del diseño a diferentes escenarios y bajo disímiles condiciones, acoplamiento entre módulos del diseño, nivel de cohesión y modularidad.

Una encuesta realizada a miembros de un equipo de desarrollo de software de la empresa venezolana *Petróleos de Venezuela S.A. (PDVSA)* y a un especialista de la empresa colombiana *Outsorcing en Desarrollos Informáticos Ltda. (ODI)* se detectó que éstos no conocían sobre alguna metodología de diseño para realizar dicho proceso en el contexto del desarrollo de sistemas informáticos. Esta encuesta además arraigó a la observación de que la estructuración organizacional de un equipo de diseñadores y la asignación de roles dentro de éste en algunos casos se realizaba basándose en los principios de la metodología de desarrollo de software que se estuviese empleando y en otros casos se realizaba de manera empírica.

Otros criterios estuvieron enfocados en la necesidad e importancia de la existencia de algún método, modelo, metodología o procedimiento capaz de brindar principios para organizar, estandarizar y controlar el proceso de diseño y que por ende favoreciera de manera positiva la calidad de diseño como proceso y como producto.

Partiendo de esto, el presente trabajo de diploma se encargará de dar solución al siguiente **problema:** *¿cómo contribuir al mejoramiento de los atributos de calidad del proceso de diseño de software?*

Por consiguiente, el **objeto de estudio** de este trabajo es el Proceso de Diseño en el Desarrollo de Software, cuyo **campo de acción** es la Calidad del Proceso de Diseño en el Desarrollo de Software.

Si se lograse definir una guía metodológica de buenas prácticas orientada a la calidad del diseño de sistemas, se podrá lograr un mejoramiento en los atributos de calidad del proceso de diseño en el desarrollo de software.

El **objetivo general** de este trabajo de diploma es desarrollar una propuesta de guía metodológica de buenas prácticas orientada a la calidad del diseño de software, que sirva de soporte a una organización o proyecto, para realizar el proceso.

Con el fin de cumplir este objetivo, se concibieron las siguientes tareas de investigación:

1. Realizar un estudio del Estado del Arte de las diferentes tendencias que abordan el Diseño en el Proceso de Desarrollo de Software.
2. Realizar un estudio del Estado del Arte de las diferentes metodologías de diseño.
3. Realizar un estudio del Estado del Arte de las diferentes Métricas de Calidad para el Proceso de Diseño de Software.
4. Realizar un estudio del Estado del Arte sobre los patrones de diseño de Software y la tendencia actual en el uso de estos dentro del Proceso de Desarrollo de Software.
5. Crear una propuesta de Guía Metodológica de buenas prácticas para el diseño de software.

6. Validar los aspectos de la Guía Metodológica y evaluarlos en un caso de estudio.

El alcance de los resultados de la investigación se refleja en la definición y aplicación de una guía metodológica de buenas prácticas para el proceso de diseño, extensible a cualquier proceso de desarrollo de software de cualquier naturaleza, bien sea en el contexto de la industria o en el de pequeños productores.

El análisis dentro del presente trabajo, en principios estará apoyado en el contexto de la Facultad 3 de la Universidad de las Ciencias Informáticas con el objetivo de poder ejemplificar sobre algunos de los aspectos que se tratarán durante todo el desarrollo de la investigación.

El trabajo consta de dos capítulos en los que se encuentra distribuido el contenido de la siguiente manera: en el primer capítulo se abordará sobre la fundamentación teórica del trabajo, la que incluye el estado del arte de las diferentes metodologías de diseño, sus principios, los atributos de calidad del diseño en cuestión y el impacto del proceso en la industria del software. En el segundo capítulo se propondrá una guía metodológica de buenas prácticas de diseño, orientada al mejoramiento de los atributos de calidad y consistente en el la propuesta de una estructura organizacional del proceso, basada en la definición de departamentos, roles de trabajo y artefactos resultantes. En este mismo capítulo se realizará una primera parte de la validación de la solución propuesta, basada en la recopilación de criterios de especialistas sobre la factibilidad de emplear dicho modelo. En trabajos posteriores se estará validando la guía mediante algún método de validación científica.

CAPÍTULO 1

Capítulo 1: FUNDAMENTACIÓN TEÓRICA

INTRODUCCIÓN

En este capítulo se muestra el estado en qué se encuentra actualmente el proceso de diseño de software en el mundo y más específicamente la visión de algunas metodologías de desarrollo de software sobre este proceso; el criterio de autores reconocidos en el ámbito y la experiencia de algunas empresas desarrolladoras de software en el mundo.

Se realiza además un análisis y evaluación de las principales problemáticas que hoy afectan el diseño de software, así como una valoración de los principios del diseño y se señalan las principales tendencias que conllevan a malas prácticas y errores en el desarrollo de sistemas. Un análisis de los atributos de calidad del diseño es otro de los aspectos a tener en cuenta en el presente capítulo; así como diferentes métodos de diseño, métricas para evaluar la calidad del diseño y el análisis del empleo de patrones, incluyendo el impacto del mal uso de estos en el proceso de desarrollo de software.

Una parte importante de este capítulo es la contextualización que se realiza del proceso así como la definición del proceso en vistas o perspectivas de diseño de software.

1.1. ¿QUÉ ES EL DISEÑO DE SOFTWARE?

El diseño es una representación ingenieril significativa de algo que se pretende construir, en este caso: un software. En principios, el diseño en el ámbito de la Ingeniería de Software como disciplina, se enmarca en cuatro áreas fundamentales: datos, arquitectura, interfaces y componentes. Se puede hacer el seguimiento de este proceso basándose en los requerimientos del cliente y al mismo tiempo, evaluar la calidad del mismo partiendo de los criterios predefinidos con el objetivos de alcanzar un *buen diseño*[1]. Se encuentra ubicado en el mismo núcleo del proceso de desarrollo de software y constituye la primera actividad técnica de las tres que se realizan en el proceso de ingeniería de software (PRESSMAN, 2001)

Las áreas en las cuales el diseño se desarrolla están estrechamente ligadas a los pasos generales o niveles de detalles que rigen el proceso de diseño: la estructuración de los elementos correspondientes al nivel de datos; la arquitectura del sistema; la representación del diseño de la interfaz y finalmente, los detalles a nivel de componentes. En cada uno de estos niveles se aplican los conceptos y principios básicos, así como patrones de diseño, que llevan a la obtención de un diseño con una alta calidad.

El diseño del software es un proceso que tributa en demasía al proceso de desarrollo de sistemas para transformar los requisitos de los usuarios o clientes en un producto de software finalizado. Este es un proceso iterativo mediante el cual los requisitos se llevan a un *plano*[2], el que estará guiando el proceso de implementación del software.

---

[1] Se puede saber si se ha desarrollado un diseño correctamente, mediante el proceso de revisión de los productos del diseño en cuanto a claridad, corrección, finalización y consistencia comparándose entonces con los requisitos del cliente y basándose evidentemente en las diferentes normas de calidad (PRESSMAN, 2001)

[2] Es una representación preliminar del software y con el empleo de elementos gráficos permitiendo un alto nivel de abstracción.

En realidad, cualquier cosa asociada a la estructura e implementación de un programa está relacionada directamente con el diseño. Esto incluye la estructura del programa, la naturaleza del lenguaje que es esté usando, el uso de los símbolos de separación en el código fuente, etc. Si bien es cierto que todos estos elementos tienen relación estrecha con el diseño, éste se desarrolla en mayor o menor grado de detalle en dependencia del contexto en que se vea (HUMPHREY, 2001).

En esencia, el diseño se puede hacer a partir de dos clasificaciones de acuerdo al nivel de detalle. Si se habla de diseño orientado a la estructuración funcional, por ejemplo, de una sentencia *case,* un *bucle* o la definición e implementación de un algoritmo recursivo, estamos en presencia de un diseño a bajo nivel. Si por el contrario el diseño está orientado a la estructuración de datos por ejemplo, o a la definición de módulos de interfaz, se está hablando de un diseño a alto nivel, asociado a un concepto que se tratará en otros epígrafes del presente trabajo: la *abstracción.*

La importancia del diseño radica en la necesidad de hacer un software más estable, con mucha más *calidad.* Precisamente es en el diseño donde se fomentará la *calidad* del producto de software final, y es donde se crearán las representaciones del software que se podrán evaluar en cuanto a calidad se refiere.

De manera que el diseño es una etapa que no deberá saltarse u omitirse en un proceso de de construcción de software. No es relevante la metodología que se emplee para desarrollar dicho software –en cuanto a diseño se refiere-, sin embargo, diseñar lo que se pretende construir, teniendo en cuenta los elementos de calidad, los patrones de diseño adecuados a emplear así como el propio método de diseño, es un imperativo en el proceso de ingeniería de software si se quiere lograr una alta calidad en el producto de software final.

1.1.1. ESTADO ACTUAL DEL DISEÑO DE SOFTWARE

En la actualidad el concepto de diseño de software está orientado a diferentes enfoques dentro del propio proceso de desarrollo de sistemas informáticos, sin embargo, existe una convergencia en cuanto a que el diseño es el móvil que dirige y lleva a cabo la transformación de los requisitos del cliente en una implementación de software.

Lo real del asunto es que no existe un método o receta que dirija este proceso de manera infalible. Grady Booch planteaba: "*el diseño es el proceso de determinar una implementación efectiva y eficiente que realice las funciones y tenga la información del análisis de dominio*", en tanto R. S. Pressman presentaba al diseño como la primera de las tres actividades técnicas del proceso de desarrollo de software: diseño, generación de código y pruebas. Otra idea que refiere Pressman consiste en la importancia del diseño del software, referente a que "*este flujo se puede describir con una sola palabra: calidad*" (PRESSMAN, 2001).

El diseño es el lugar en donde se fomentan la calidad en la ingeniería del software. Éste proporciona las representaciones del software que se pueden evaluar en cuanto a calidad, sirve como fundamento para todos los pasos siguientes del soporte del software y de la ingeniería del software, por lo que sin un diseño se corre el riesgo de construir un sistema inestable, un sistema que falla cuando se lleven a cabo cambios, un sistema que puede resultar difícil de comprobar y un sistema cuya calidad no puede evaluarse hasta muy avanzado el proceso, sin tiempo suficiente y con mucho dinero gastado.

El proceso de diseño aparece explicitado en todas las metodologías de desarrollos de software en alguna medida. El mismo es ubicado o contextualizado siempre luego de alguna etapa de análisis y anterior a la actividad de implementación en cuestión.

1.1.2. EVOLUCIÓN DEL DISEÑO

La evolución del diseño de software, como parte del proceso de desarrollo de software, es un proceso continuo que se ha ido produciendo durante las últimas tres décadas. Los primeros trabajos sobre diseño se centraron sobre los criterios para el desarrollo de programas modulares y los métodos para mejorar la arquitectura del software de una manera descendente. De hecho, la modularidad se ha convertido en un enfoque aceptado en todas las disciplinas de ingeniería. Un diseño modular reduce la complejidad, facilita los cambios y da como resultado una implementación más fácil al fomentar el desarrollo paralelo de las diferentes partes de un sistema.

Los aspectos procedimentales de la definición del diseño evolucionaron hacia una filosofía denominada *programación estructurada*.

Posteriores trabajos propusieron métodos para la traducción del flujo de datos o de la estructura de los datos, en una definición de diseño. Nuevos enfoques para el diseño proponen un método orientado a objetos para la obtención del diseño.

Cada metodología de diseño de software introduce heurísticas y notaciones propias, así como una visión algo particular de lo que caracteriza a la calidad del diseño. Sin embargo, todas las metodologías tienen un conjunto de características comunes: algún mecanismo para la traducción de la representación del campo de información en una representación de diseño (mecanismos de capturas de requerimientos); una notación para representar los componentes funcionales y sus interfaces (lenguajes de modelado); heurísticas para el refinamiento y la partición (patrones de diseño); y criterios para la valoración de la calidad (atributos de calidad).

1.1.3. EL PROCESO DE DISEÑO ORIENTADO A OBJETOS (DOO)

El término *diseño* tiene disímiles acepciones de acuerdo al contexto en que se pretenda definir. Hace referencia a la actividad de representar gráficamente una

indicación de sentido o dirección, ya sea en soporte analógico, digital o virtual, y en dos o más dimensiones. Es el proceso previo de configuración (pre-figuración) mental en la búsqueda de una solución en cualquier campo. Sin embargo, diseñar, en el ámbito de la Ingeniería de Software es más que hacer representaciones gráficas sobre cualquiera de los ya antes mencionados soportes.

El diseño forma parte importante del Proceso de Desarrollo de Software sin importar cuál sea la metodología en cuestión que se pretenda seguir. Muy a pesar de que éste define en sí mismo una serie de etapas o pasos a seguir para definir una estructura preliminar y lista para hacer un desarrollo "físico" del software, trae consigo una serie de principios o fases que son inherentes a su propia naturaleza.

La actividad de diseñar, en cualquiera de los casos, debe partir de la acción de las siguientes funciones que a continuación se listan:

1. Primero, parte de la ***observación y análisis*** del medio en el cual el ser humano se desenvuelve con el objetivo de descubrir alguna necesidad de acuerdo a sus funciones en dicho entorno.
2. Una vez detectada la necesidad, se procede a la ***planeación y/o proyección*** en el que se propone un modo de solucionar la necesidad, mediante modelos o maquetas que recreen de manera visual las posibles soluciones para viabilizar el análisis de lo que sería más óptimo hacer.
3. La ***construcción y ejecución*** vendría siendo uno de los duetos más importantes dentro de este proceso de diseño, en conjunto con el paso anterior. Se encarga de materializar lo ya antes modelado. A partir de aquí, se obtiene un producto o más bien un resultado, listo para ser evaluado en cuanto a su efectividad en la solución del problema que animó la realización del proceso, evidentemente, la necesidad detectada.
4. Como se había mencionado, ***evaluar*** lo que se obtiene de la fase de construcción y ejecución, permite que se determine cuándo se ha finalizado y alcanzado una solución que si bien no es la más óptima, al menos da

respuesta al objetivo de alcanzar una solución que resuelva la necesidad inicial.

Es preciso destacar que entre estas fases o etapas existe una interacción consistente en la retroalimentación. Quizás sea preciso ver este proceso no desde un punto de vista jerárquico sino iterativo, cuya interacción sea matricial y no secuencial, partiendo de la idea de que cada una de estas fases aparece una y otra vez en la actividad de diseñar.

Para el caso de la Ingeniería de Software, el diseño no está exento de transitar por estas fases, sin embargo, se encuentra enriquecido por etapas propias del mismo y adaptadas a las características de lo que se construye: *software*.

El diseño orientado a objetos (DOO) cumple una serie de requisitos que se concretan en la necesidad de la existencia de una arquitectura de software, en la que se especifiquen subsistemas que realicen funciones y provean soporte de infraestructura de objetos (clases), los que constituyen los bloques de construcción del sistema; así como una descripción de los mecanismos de comunicación que permitan que los datos fluyan.

La principal virtud que tiene el diseño orientado a objeto radica en su capacidad de definir y desarrollar cuatro conceptos que hacen atractivo y más robusto a los sistemas orientados a objetos. Estos conceptos, entre otros, son la abstracción, ocultamiento (u ocultación) de la información, modularidad e independencia funcional. Ellos son alcanzados únicamente usando un diseño orientado a objetos, siendo proveído por este de un modo sencillo y sin compromiso. Muy a pesar de que todos los métodos tratan de exhibir estas características, solo el diseño orientado a objeto brinda un mecanismo capaz de desarrollarlas al máximo.

Pressman define cuatro capas del proceso de diseño orientado a objetos, reflejadas en lo que él nombró *la pirámide del diseño orientado a objetos* (PRESSMAN, 2001), como se muestra en la figura 1.1.1.1.

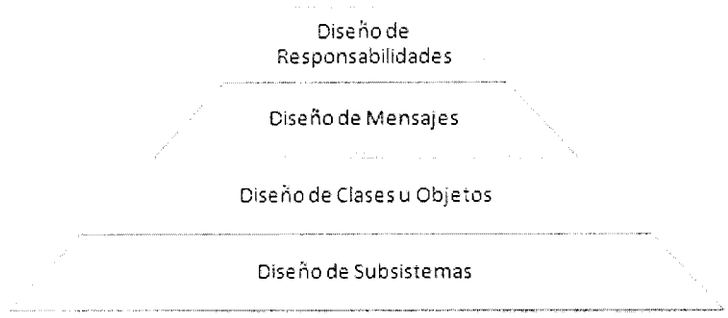

Fig. 1.1.1.1 La Pirámide del Diseño Orientado a Objetos (DOO) (PRESSMAN, 2001)

En la capa de **Diseño de Subsistemas** está contenida la representación de todos los subsistemas que permiten al software definir una infraestructura que soporte todos los requerimientos de los clientes.

La jerarquía de clases y las relaciones entre éstas que permiten la creación concreta del sistema, se enmarca en la capa de **Diseño de Clases y Objetos.**

En la capa de **Diseño de Mensajes** se definen interfaces internas y externas para que los objetos del sistema se comuniquen, así como los detalles de diseño de cómo cada uno de estos objetos se va a comunicar con sus colaboradores y es manejado por el modelo objeto-relación.

Y finalmente la capa de **Diseño de Responsabilidades** es donde se define toda la estructura de datos y diseños algorítmicos de todos los atributos y operaciones de cada objeto.

En resumen, el diseño orientado a objetos es un proceso que se aplica únicamente a sistemas orientado a objetos, brindando un mecanismo sencillo para desarrollo de conceptos como la abstracción, el ocultamiento (u ocultación) de la información, modularidad e independencia funcional. Este a su vez, se centra en definir los procedimientos que dan "vida" a la arquitectura del software en construcción.

1.2. EL DISEÑO DENTRO DEL PROCESO DE DESARROLLO DE SOFTWARE

El diseño deviene en uno de los principales procesos dentro del ciclo de vida del desarrollo de un software. Sin embargo, si bien es cierto que el proceso de diseñar antes lo que se pretende construir después es una necesidad para lograr el cumplimiento de los objetivos planteados, que en el caso del desarrollo de software no es más que lograr ajustar el sistema a los requisitos del cliente, este proceso en sí no está desligado de otros como lo es el proceso de análisis o el de implementación. Es decir, muchos autores establecen una relación entre las actividades de analizar y diseñar de manera que es imposible definir dónde termina una y comienza la otra.

Pressman define el modelo de diseño partiendo de la conversión de los aspectos del modelo de análisis y estructura el flujo de información de éste de la siguiente manera (PRESSMAN, 2001):

- Diseño de Datos
- Diseño Arquitectónico
- Diseño de Interfaz
- Diseño a nivel de componente

Cada uno de los métodos existentes ubica el diseño en un marco específico dentro del proceso de ingeniería en cuestión. En este epígrafe, se presentará una evaluación de algunos de estos métodos y cómo cada uno de ellos define el diseño de software y en qué contexto lo enmarca.

1.2.1. EL DISEÑO SEGÚN RATIONAL UNIFIED PROCESS (RUP)

RUP (del Inglés *Rational Unified Process*) plantea un conjunto de flujos de trabajo dentro del proceso de desarrollo de software, en el que el diseño queda enmarcado de manera acentuada en un flujo de trabajo denominado **Análisis y Diseño**.

Graig Larman en su libro "*UML y Patrones*" nos plantea una transición entre el análisis y el diseño en cuestión (LARMAN, 2003). Básicamente esto apoya la idea expresada anteriormente, a inicios del epígrafe 1.2.1, sobre lo que se podría llamar "fases elementales del proceso de diseño", y en el que evidentemente las actividades de analizar y diseñar propiamente dichas, están delimitadas por una fase en la que se obtienen una serie de componentes (entiéndase componentes como artefactos) que marcan la frontera entre un proceso y otro. Es decir, en el caso de RUP, una vez que se obtienen una serie de artefactos como son los Casos de Uso, el Modelo Conceptual, Diagramas de Secuencia del Sistema, Contratos y otros más, se está en condiciones de pasar a la siguiente etapa, que sería el diseño, una vez terminados estos documentos del análisis. Durante este paso de transición se logra obtener una solución lógica del problema, la que en esencia se enmarca partiendo de los principios de la orientación a objeto que caracteriza el proceso de RUP, en los diagramas de interacción.

De acuerdo con los principios de RUP como metodología, el diseño de software tiene diferentes funciones esenciales (JACOBSON, y otros, 2000):

1. Tener una comprensión en profundidad de los requisitos funcionales y no funcionales obtenidos de fases anteriores al diseño, así como de las diferentes tecnologías propuestas, las restricciones de los lenguajes de programación, etc.

2. Definir elementos que sirvan para la actividad de implementación subsiguiente.

3. Descomponer la implementación en partes menos complejas como subsistemas, módulos, componentes de clases, etc.

4. Capturar las interfaces de los subsistemas antes del ciclo de vida del software.

5. Crear una abstracción de la implementación del sistema sin caer en especificidades, en el sentido de que la implementación es el refinamiento del diseño que se encarga de rellenar lo que ya está diseñado sin cambiar la

estructura.

Básicamente, mediante el diseño se obtienen una serie de artefactos que tributan con elementos claves a la actividad de implementación. Ambos procesos, tanto el diseño como la implementación, guardan entre sí una estrecha relación reflejada en la continuidad que ejerce la implementación sobre los elementos diseñados hasta el punto de usar tecnologías de generación de código a partir del diseño propiamente dicho, y la posibilidad, usando herramientas de ingeniería inversa, de llegar a un modelo de diseño partiendo de la implementación en cuestión. A esto se le debe sumar que existe una retroalimentación entre los procesos de diseño e implementación.

RUP propone nueve flujos de trabajo y cuatro fases de desarrollo (véase Fig. 1.2.1.1) en las que el diseño constituye uno de estos flujos de trabajo y comienza a cobrar mayor importancia dentro del proceso a la hora en punto en que se inicia la fase de Elaboración, permitiendo enriquecer la arquitectura y hacerla más estable y sólida, además de definir un plano lo suficientemente completo como para iniciar la tarea de implementación a mayor escala en el momento en que se transita de la fase de Elaboración hacia la fase de Construcción, esta última en la que la implementación comienza a cobrar mayor atención.

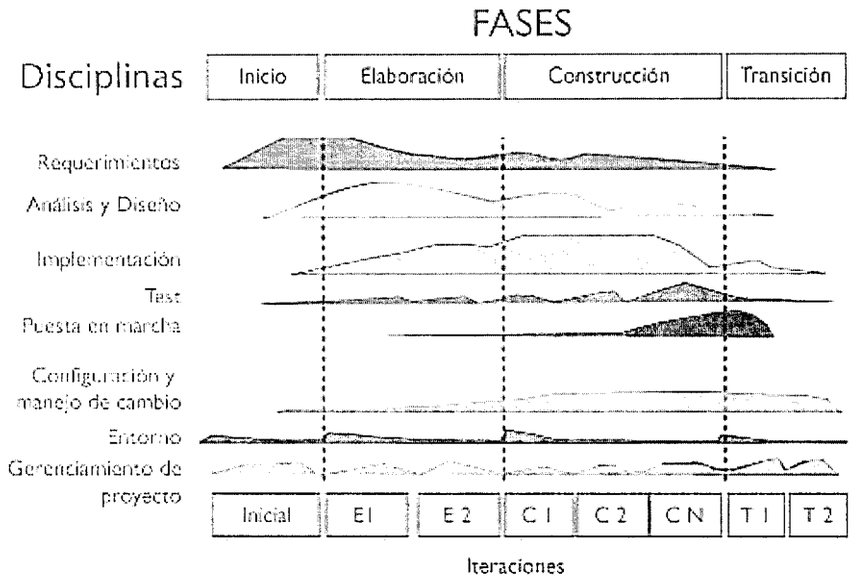

Fig. 1.2.1.1: Flujos de Trabajo y Fases de Desarrollo de RUP

1.2.2. EL DISEÑO SEGÚN SCRUM

ESCRUM es una metodología ágil de desarrollo de software con un alto grado de solapamiento entre las actividades provocando que no existan fases de de desarrollo sino tareas que se asignan y se ejecutan cuando sea necesario (PALACIO BAÑARES, 2007).

Esta metodología incentiva la participación colectiva de todo el equipo en el diseño como manera de aportar más conocimiento innovador y diferencial, además de esto, estimula la práctica del intercambio cruzado de conocimiento o lo que se denomina "*fertilización cruzada*" entre equipos *auto-organizados* producida por la "*ingeniería concurrente*" consistente en el solapamiento de las fases, logrando mejores resultados que de manera aislada.

En realidad, SCRUM es una metodología que está orientada a la gestión de proyectos de software, en la cual el diseño se construye de manera evolutiva debido a la inestabilidad del entorno y de los requisitos.

En la aplicación de SRUM para software, para evitar los problemas de degradación del sistema o de la arquitectura por la evolución continua del producto se deben incluir prácticas de refactorización en las tareas de diseño e implementación.

1.2.3. EL DISEÑO SEGÚN EXTREME PROGRAMMING (XP)

Por otra parte, XP presenta dentro de sus prácticas la definición de un diseño simple capaz de ser implementado en un momento determinado del proyecto. Ahora, lo curioso es que de acuerdo con la propuesta original de Beck para la especificación de los roles que participan en el proceso de desarrollo según XP, no existe un diseñador de software como tal, por lo que se asigna esta responsabilidad al propio rol de programador, el que además debe escribir las pruebas unitarias y producir el código del sistema. Para este caso, y respondiendo precisamente a otra de las prácticas de XP –*la programación en parejas*-, la actividad de diseñar lo que se implementa, recae sobre la pareja de programadores que desarrolla dicha implementación (CANÓS, y otros, 2003).

Como se había expresado en ocasiones anteriores, se le confía la responsabilidad de diseñar a la capacidad o experiencia del propio programador. Muy a pesar de esto, no se excluye el uso de algún método en específico de diseño, así como también de buenas prácticas como lo es precisamente el propio empleo de patrones de diseño.

Sin embargo, este proceso de diseñar y programar en parejas no es tan simple como parece. A esto se le debe sumar que debido a que el código es público, atendiendo a otra práctica de XP –*la propiedad colectiva del código*-, hace que otros programadores puedan revisar el código y efectuar cambios en él. Para modificar el

código fuente es preciso modificar o al menos analizar el diseño definido, haciendo necesario la existencia de estándares para un mejor entendimiento del código. A menos que los cambios en el código sean muy sencillos, el analizar el diseño es un paso de relevante connotación y que no debería saltarse para ganar en tiempo de entendimiento y familiarización.

Partiendo de todo lo anteriormente planteado, puede arribarse a la conclusión de que no es posible garantizar una buena calidad del diseño, no solo como proceso sino también como producto, a menos que éste se institucionalice o lo que es lo mismo, se defina un estándar que permita "*hablar*" el mismo idioma entre programadores a nivel de diseño, facilitando aún más la comunicación entre ellos.

En realidad XP plantea el uso de estándares, pero de programación, partiendo del principio de que la principal forma de los programadores comunicarse es el propio código. En cambio, el empleo de *estándares* para el proceso de diseño, o lo que podría ser una guía metodológica para llevar a cabo el proceso de diseño, sería de muy buena aceptación debido a que enriquecería dicha comunicación entre programadores y por tanto, permitiría obtener un producto de mejor calidad. Sería muy conveniente además, crear algún mecanismo de documentación para este proceso, el que sin complejizar la construcción del sistema, permita persistir los modelos de diseño.

1.2.4. EL MÉTODO DE BOOCH

El método de Booch define un proceso de micro desarrollo y macro desarrollo, ambas partes incluyen varios pasos como son la identificación de clases y objetos a un nivel de abstracción dado, la identificación de la semántica de esas clases y objetos, la identificación de las relaciones entre esas clases y objetos, la selección de la estructura de datos y algoritmos para la implantación de estas clases y objetos, la conceptualización del sistema, etc. En el contexto del diseño, el macro desarrollo crea

un prototipo de diseño y valida dicho prototipo aplicándolo a situaciones de uso. Además, engloba una actividad de planificación arquitectónica que se encargan de agrupar objetos similares en particiones arquitectónicas separadas, define capas de objetos por nivel de abstracción e identifica situaciones relevantes.

En el caso del micro proceso, se encarga de definir un conjunto de reglas que delimitan el uso de atributos y operaciones, así como las políticas de dominio para la administración de memoria, manejo de errores y otras funciones (PRESSMAN, 2001).

1.2.5. EL MÉTODO DE RUMBAUGH

Rumbaugh define lo que se conoce como *Técnica de Modelado de Objetos (OMT)*, consistente en una metodología que pretende ir añadiéndole detalles de implementación durante el diseño del sistema al modelo de dominio de aplicación.

Consta de tres tipos de modelos diferentes: *Modelo de Objetos,* el que se encarga de describir la estructura estática y las relaciones de los objetos el sistema; *Modelo Dinámico,* se encarga de implementar los aspectos de control del sistema representados mediante diagramas de estados; y *Modelo Funcional,* describe las transformaciones de valores de datos, representados en diagramas de flujos de datos del sistema.

Este método centra el diseño del sistema en el esquema de los componentes que se necesitan para construir el producto de software.

El diseño a nivel de objeto enfatiza los detalles de cada objeto individual, se seleccionan las operaciones que define el modelo de análisis y se representan las estructuras de datos apropiadas para atributos y algoritmos.

En este método las clases son diseñadas de manera que se optimice el acceso a los datos con el objetivo de mejorar la eficiencia computacional. Las relaciones de objetos (clases) se establecen mediante un modelo de mensajería, el cual

implementa las asociaciones y la manera en que colaboran los objetos (clases) (PRESSMAN, 2001).

1.2.6. EL MÉTODO DE JACOBSON

El diseño de software orientado a objetos en el marco de la Ingeniería de Software, es una versión simplificada del método propietario *Objectory,* creado también por *Jacobson.*

En principios este método, introduce el concepto de *bloque* como la abstracción de diseño que permite la representación de un objeto agregado. Estos bloques no son más que los objetos de diseño primarios que posteriormente son catalogados como bloques de interfaz, bloques de entidades y bloques de control. Finalmente la comunicación entre estos bloques es definida durante la ejecución del diseño y finalmente agrupados en subsistemas (PRESSMAN, 2001).

1.2.7. EL MÉTODO DE COAD Y YOURDON

Este método es producto de la experiencia del trabajo de diseñadores en el proceso de diseño orientado a objetos. Expone al diseño no solo en su orientación a la aplicación, sino también a su infraestructura, y se enfoca en cuatro componentes mayores de sistemas: una componente de dominio del problema, una componente dirigida a la interacción humana, la componente de administración de tareas y de administración de datos (PRESSMAN, 2001).

1.2.8. EL MÉTODO DE WIRFS-BROCK

Este método, como otros, define un conjunto de tareas técnicas que marcan el hilo conductor del análisis al diseño.

Los *protocolos*[3] para cada una de las clases se definen refinando contratos entre objetos. Los protocolos están asociados al diseño de interfaces de clases.

Cada una de las responsabilidades y las interfaces se diseñan hasta un nivel de detalle necesario para guiar la implementación.

1.3. CONTEXTUALIZACIÓN DEL DISEÑO

Sin ánimo de entrar en definiciones sobre lo qué es diseño, es preciso establecer definitivamente dónde es que se encuentra ubicado el diseño dentro del proceso de desarrollo de software, pero más que esto, analizar cuál es la relación exacta que existe entre éste y la tecnología empleada.

Sobre esto último queda claro que el diseño tiene una estrecha relación con la tecnología, puesto a que es un modelo físico no genérico y específico para una implementación de un lenguaje en determinado (JACOBSON, y otros, 2000). Sin embargo, la relación hay que verla en dos direcciones; faltaría analizar qué influencia tiene la tecnología en el desarrollo del diseño y qué limitantes le impone asociadas a los conceptos de portabilidad, uso de patrones y otros elementos que pueden ser especificados por el propio usuario o por el momento en que se desarrolle el diseño.

Muy independientemente de la metodología de software que se emplee, el diseño siempre se ubica luego de un proceso de análisis y anterior al proceso de implementación. Sin embargo, el ser precisamente la antesala de la implementación del producto de software, no puede interpretarse que es el diseño quien determina la tecnología que se va a usar, sino esta última quien limita el desarrollo de los conceptos del diseño antes mencionados, asociado más específicamente al empleo de patrones de diseño.

---

[3] Se define *protocolo* como la descripción formal de los mensajes a los que la clase responde.

Muchos autores contextualizan el diseño basado en su posición entre el análisis y la implementación, sirviendo de comunicación de ambos procesos, y especificando la dependencia con la tecnología en cuanto a la capacidad de éste de visualizar la implementación mediante las técnicas de programación gráficas (JACOBSON, y otros, 2000). Sin embargo, existe otro elemento a tener en cuenta para una completa contextualización: el momento histórico en que se enmarque el proceso.

El momento histórico determina las tecnologías imperantes y los paradigmas más usados.

Para que se comprenda mejor lo anteriormente explicado, se presenta la siguiente imagen (Fig. 1.2.7.1) que pretende recrear en un ambiente gráfico dónde se ubica el diseño en el proceso de desarrollo software y qué factores inciden en él.

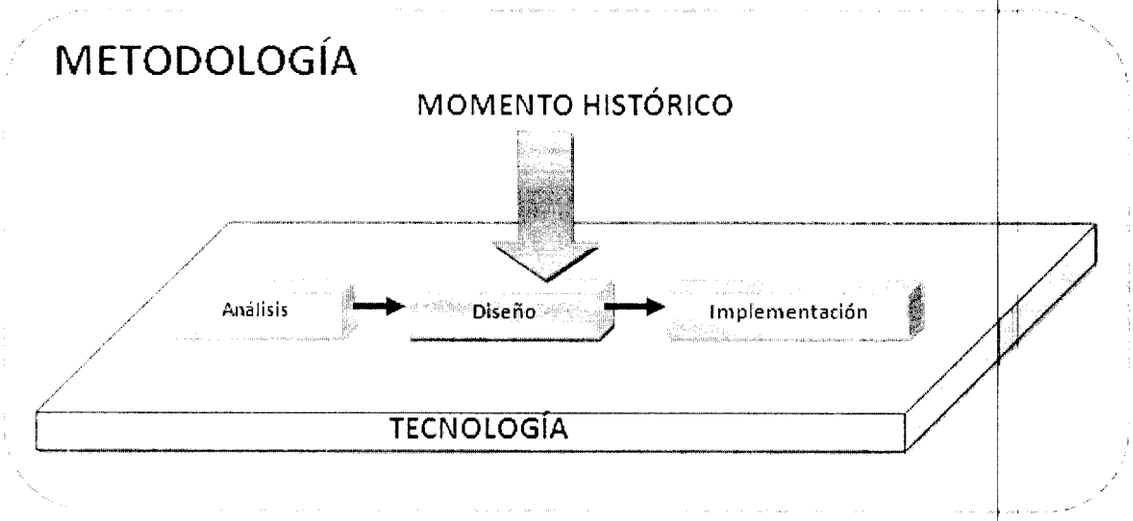

Fig. 1.2.7.1 Contextualización del Diseño en el Proceso de Desarrollo de Software

1.3.1. VISTAS DEL PROCESO DE DISEÑO

Es conveniente concebir el diseño desde diferentes perspectivas. Ya anteriormente se abordaba sobre las áreas en las que el diseño se enmarca: datos, arquitectura, interfaces y componentes.

Fig. 1.2.7.2 Vista del Proceso de Diseño de Software en perspectivas de diseño.

Sin embargo, la idea no es ver al diseño por partes sino variar el objeto de análisis de acuerdo a la perspectiva de diseño en la cual se centre dicho análisis.

En la perspectiva del marco tecnológico, se realiza un análisis exhaustivo sobre la incidencia de la tecnología a emplear para implementar el sistema, partiendo de una arquitectura previamente definida, y de los requisitos no funcionales especificados por el cliente.

En el caso de la perspectiva de interrelación de elementos de software, el punto de atención estaría dirigido a definir en el diseño la manera en que los objetos del sistemas, las clases, interfaces, etc., se relacionan y/o colaboran entre sí, los mensajes que intercambian, la manera de responder a estos mensajes y la aplicación de conceptos asociados al paradigma que se esté empleando en el momento en que se desarrolla el diseño. Además de realizar un estudio de los patrones de diseño más adecuados para lograr un alto grado de modularidad, reutilización e independencia funcional.

Para la perspectiva de rendimiento y seguridad, se centrará el estudio en los temas de programación segura y los temas de rendimiento del sistema, tal y como lo sugiere el propio nombre de la perspectiva. Para el rendimiento, el tratamiento del tipiado de datos, el uso de elementos externos de la aplicación (como ficheros XML, accesos a bases de datos para *cargar* alguna información, etc.), el empleo de las diferentes estructuras de datos, entre otros elementos, deberán tenerse en cuenta para lograr que el futuro software funcione de manera eficiente y de acuerdo a los tiempos de respuestas mínimos necesarios. A todo esto se le suma, para el caso de la seguridad, la no sobrecarga de *buffers*, que los *bucles* sean finitos, que se garantice los diferentes niveles de seguridad, entre otros aspectos.

En la perspectiva de interfaz se definiría la extensibilidad del diseño. En la cual sería la encargada de brindar las capacidades del producto de software para su posterior integración con otros sistemas. El diseño debe ser lo suficientemente flexible y extensible como para poder permitir integrar otros componentes de software sin necesidad de transformar toda la concepción del mismo.

Hasta este momento se ha podido analizar los principales criterios planteados por autores, o los expuestos por algunas metodologías de desarrollo de software e incluso empresas de la industria del software a cerca de la importancia, la contextualización y la responsabilidad del diseño de software. Teniendo en cuenta todo esto, se puede llegar a la siguiente conceptualización de lo que es el diseño de

software: *es un proceso de modelado para abstraer procesos intrínsecos del negocio, subyacentes en el problema a resolver y enmarcados en la propia realidad concreta, expresados en las necesidades del cliente. Se contextualiza principalmente en cuatro perspectivas de diseño: marco tecnológico, interrelación entre elementos de software, rendimiento e interfaz y puede ser expresado en dos categorías de acuerdo al nivel de abstracción: diseño de alto nivel y diseño de bajo nivel.*

1.4. PRINCIPALES CAUSAS DE PROBLEMAS DEL DISEÑO DE SOFTWARE

Los problemas que a menudo se comenten en el proceso de diseño de software, están asociados a los atributos de calidad del propio proceso en cuestión. Sin embargo, son precisamente los problemas del diseño los que tipifican los atributos de calidad del mismo y no estos los que determinan los problemas de diseño; es decir, dan una medida de qué es lo que se debe medir y cuáles son los niveles aceptables de la solución al problema.

Anteriormente se mencionaban algunos de los principales problemas del diseño y su impacto negativo en el proceso en cuestión. La **poca flexibilidad y extensibilidad**, el **alto acoplamiento** y la **baja cohesión** son algunos de estos, pero no los únicos. La tendencia que existe hacia el *rediseño* como solución a las consecuencias de cambios en los requisitos, así como la rigidez, la fragilidad, la inmovilidad y la viscosidad del diseño, son algunos de otros problemas de significativa relevancia, siendo estos cuatro últimos más que problemas, síntomas de que se ha realizado un mal diseño.

Al principio los problemas en el diseño son prácticamente imperceptibles, pero con el tiempo y debido a la cantidad de "*parchos*" que se le van agregando, éste se vuelve absolutamente difícil de mantener. Un simple análisis conlleva a grandes esfuerzos para hacer el más sencillo de los cambios, por lo que la solución más "adecuada" es la de rediseñar.

Los rediseños generalmente no funcionan. Muy a pesar de que el diseño no pierde su filosofía, el sistema va evolucionando con el tiempo y nuevamente vuelven a cometerse nuevos errores y el rediseño nunca termina (GRACIA, 2004).

La rigidez es un concepto análogo a la poca flexibilidad del diseño, muy asociado a la dependencia funcional debido a que los cambios en un módulo provocan cambios en otros módulos independientes, haciendo que el software sea *frágil* ante la más mínima modificación.

En el caso de la inmovilidad, el diseño tiende a ejercer una resistencia a ser reutilizado en otros sistemas u otras partes del sistema.

Finalmente, la viscosidad está expresada en la dificultad de mantener la filosofía del diseño original. En ocasiones los programadores encuentran muchas formas de implementar un cambio. Cuando esa implementación es muy compleja para preservar la filosofía del diseño, puede decirse que se está en presencia de un alto nivel de *viscosidad*.

Existen varias causas que determinan estos problemas. Primero, la poca o nula comprensión de de lo que debe hacer el sistema, como por ejemplo alguna función matemática no comprendida o el ignorar alguna condición del sistema, provocaría un *diseño erróneo* (HUMPHREY, 2001).

Una segunda causa sería, conociendo lo que se debía hacer, se comete un simple error de descuido en cuanto a la completitud de la funcionalidad a modelar; ejemplo la no incorporación de todos los casos a evaluar en alguna sentencia de casos, provocaría un *diseño incorrecto*.

La tercera causa a evaluar sería la de tergiversar lo que se quiere hacer, tanto si se ha mal interpretado el diseño de alto nivel o si no se han entendido los requisitos del sistema, provocando un *diseño incorrecto* aunque para los efecto el diseño estaría acorde a lo que se entendió o se creyó entender.

Otra causa importante y muy asociada a la tercera causa analizada, es la de concebir de manera correcta el diseño pero realizar una mala representación del mismo, es decir, el diseño fue adecuadamente concebido pero pobremente representado.

Estas son algunos de los principales problemas asociados al proceso de diseño y sus respectivas causas que los provocan.

1.4.1. PRINCIPALES PROBLEMAS DE LA CALIDAD DEL DISEÑO DE SOFTWARE

La calidad del diseño se realiza durante todo el ciclo de vida del proceso y mediante un conjunto de tareas que se concretan en la realización de un enfoque de gestión de la calidad; el empleo de tecnologías de ingeniería de software efectivas usando métodos y herramientas para ello; revisiones técnicas formales que se aplican durante todo el proceso; una estrategia de prueba *multiescalada*; el control de la documentación del diseño, que incluye además la de los cambios realizados; un procedimiento que asegure un ajuste y buen uso de los estándares internacionales y mecanismos de medición y generación de informes.

La calidad algunos la definen como "*una característica o atributo de algo*" (*American Heritage Dictionary*) o como "*el grado que el software tiene de una combinación deseada de los atributos (modificación, seguridad, rendimiento, disponibilidad, etc.)*" (BARBACCI, 2005). Lo cierto es que la calidad es una actividad de protección que se aplica a lo largo de todo el proceso de desarrollo del software (PRESSMAN, 2001).

Pressman define dos tipos de calidad de acuerdo a las características mesurables de un software: la calidad del diseño y la calidad de concordancia. La primera se refiere a las características que especifican los ingenieros de software para un elemento, de manera que la calidad del diseño aumenta si el producto se fabrica de acuerdo con las especificaciones. La calidad de concordancia está asociada a la actividad de

implementación y al cumplimiento con las especificaciones de diseño (PRESSMAN, 2001).

Uno de los principales problemas en la calidad del diseño es que hay una mala interpretación de cómo obtener una mejor calidad. Primero que existe una tendencia a enfocarse en la reducción de tiempo y costes de desarrollo cuando en realidad debiera ser en sentido contrario, *centrarse en la calidad para poder alcanzar una verdadera reducción a largo plazo de costes y tiempo de desarrollo*. Precisamente este enfoque –o mejor dicho, este mal enfoque- se aplica a muchos proyectos que se desarrollan en la Facultad 3, provocando una incidencia negativa en los atributos de calidad del diseño.

Por otra parte están las revisiones técnicas formales. Estas constituyen un mecanismo esencial para la detección de errores, cuyo principal objetivo es la no propagación de estos errores a una etapa siguiente en el desarrollo de software, evitando de esta manera que se conviertan en defectos luego de entregado el producto. Estudios realizados por la *Nippon Electric y Mitre Corp.* entre otros, plantean que de un 50 a un 65 por ciento de los errores (y en última instancia, los defectos), son introducidos por la actividad de diseñar. Por esta razón es preciso la realización de revisiones formales y más aún cuando su efectividad oscila entre un 70 y un 100 por ciento en la detección de errores.

Aunque en un momento posterior se tratará el tema del uso de los patrones de diseño, es preciso mencionar que otro de los problemas en la calidad de un diseño, es el uso inadecuado y en muchos casos nulo, de estos patrones. Es importante aclarar que en tanto es una mala práctica el no usar patrones de diseño donde se deberían usar, es totalmente ineficiente, además de que los costes por refactorización y mantenimiento aumentan, el uso inadecuado de estos patrones, es decir, usar un patrón X dónde debería ir un patrón Y, podría traer consigo una elevada cohesión provocada por las dependencias indeseables creadas por patrones híbridos (PÉREZ

MIRIÑÁN, 2002). A todo este proceso de mal uso de patrones de diseño se le conoce como sobre-ingeniería (del Inglés *overengineering*).

Por último, la violación del principio de correspondencia con los requerimientos del sistema hace que el diseño esté expuesto constantemente a cambios, en principios innecesarios, cayendo en lo que se conoce como *rediseño* –mencionado anteriormente- . Evaluar dicha correspondencia desde inicios del proceso, por cualquiera de los mecanismos antes mencionados, evita que se caiga en tan ineficiente práctica.

1.4.2. ATRIBUTOS DE CALIDAD DEL DISEÑO DE SOFTWARE

De acuerdo a la norma ISO 9126 (ISO/IEC, 2001) la calidad de un producto de software puede clasificarse de acuerdo al contexto, en *calidad interna* o *calidad externa*. Ésta debe ser evaluada sobre la base de los atributos de calidad, los que de manera análoga también pueden ser *internos* (propiedades o características de cómo se estructura el software) o *externos* (cualidades observables aún sin conocer cómo está construido el software). El desarrollo del presente trabajo se estará haciendo alusión a los atributos de calidad del software, pero más específicamente en el marco del proceso de diseño.

La propia norma ISO 9126 estable seis características generales para definir la calidad del software, bien sea la calidad externa como la interna (*funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad*) (MARÍN, y otros, 2007). Es evidente que no todas deberán ser medidas en el diseño de software, no solo porque éste sea un proceso meramente interno y no sería posible evaluar característica externas de calidad, sino porque hay incluso algunas de estas seis características que son internas, pero aún así no corresponden al proceso de diseño en cuestión.

Es válido aclarar, que muy a pesar que se tipifiquen estos dos grandes grupos, no es conveniente verlos de manera definitivamente desligada ya que ambas clasificaciones se complementan para permitir evaluar la *calidad total del producto de software*[4], además de esto, la calidad interna influye de manera directa en la calidad externa. Es decir, mejorando la calidad interna, se mejora la calidad observada en atributos de uso del software (OLMEDILLA ARREGUI, 2005).

Por lo que respecta a la ***funcionalidad***, la medición de esta corresponde a la verificación de la correcta y completa cobertura de los requisitos de los usuarios, por tanto, se está haciendo alusión de algo que debiera hacerse durante una etapa de análisis y evaluado mediante pruebas de cobertura en etapas posteriores al diseño, como es la implementación, cuando ya se tiene el código. Respecto a esto, la ISO 9126 plantea que la calidad del diseño no debiera evaluarse por la funcionalidad del diseño. Un ejemplo que puede ilustrar mejor este criterio, es en el caso en que existan dos diseños que respondan a los mismos requisitos de usuarios y uno de estos no se ajusta correctamente o de manera completa, no podría decirse que ese es "*peor*" que el otro, sino que simplemente es incorrecto o incompleto.

Sucede similar para el caso de la ***fiabilidad***, ya que esta es una característica que se mide durante la etapa de prueba de un sistema, no debería considerarse uno de los atributos internos que definen la calidad del diseño. Muy a pesar de esto, la complejidad ciclomática es un mecanismo que pretender resolver este problema, logrando definir el número mínimos de pruebas que son necesarias para determinar un nivel de cobertura de los requisitos de usuarios y por ende el número de defectos latentes. Desde hace más de una década, se han definidos métricas para el software orientado a objetos, que tratan de predecir la fiabilidad desde la propia etapa del diseño (OLMEDILLA ARREGUI, 2005).

---

[4] Entiéndase como *calidad total del producto de software* como a la suma de analizar la calidad interna y la calidad externa del producto de software.

La ***usabilidad,*** que es otra de las características generales que define la ISO 9126, está directamente relacionada con la forma en que el usuario percibe el producto terminado. En cambio, se le podría dar otro enfoque a esta característica para ajustarla al marco del diseño. Para ello, sería preciso presentar al *diseñador* como un usuario del diseño. Por esta razón, se podría proponer la *comprensibilidad* como un posible atributo interno del diseño, lo que haría pensarse que se tendría el problema resuelto, mas no sería del todo así. Existe ya un atributo de calidad interna del diseño que abarca todo lo relacionado al nivel de comprensibilidad, la facilidad de análisis y mantenimiento del diseño y es precisamente la ***mantenibilidad***.

La ***eficiencia,*** según la ISO 9126, se divide en *comportamiento temporal* y *utilización de recursos,* lo que evidentemente ha hecho de esta característica un claro objetivo de medición de la etapa de prueba de un software. En cambio, existe una tendencia en la actualidad a medir la eficiencia en el diseño, partiendo de una característica que sí podría considerarse interna del diseño: el *rendimiento*. Existen criterios que contraponen el *rendimiento* de un diseño y su nivel de *comprensibilidad* debido que para lograr un alto grado de este último, se emplean patrones de diseño, los que en ocasiones penalizan el rendimiento del sistema. Sin embargo, se debería ver desde otra perspectiva. El empleo de patrones además de facilitar la comprensibilidad del diseño –entre otras muchas cosas más-, permite aislar aquellos "*focos de bajo rendimiento*" para darle un tratamiento más especial.

Por último, la ***portabilidad*** a todas luces parece estar fuera del alcance del proceso de diseño. Éste bien podría considerarse un atributo poco importante para el caso del diseño, a lo que a calidad se refiere, ya que el diseño en sí debe mantenerse conceptualmente separado del entorno de implementación final. Sin embargo, si se tuviese en consideración que el diseño ha de tener en cuenta el entorno final de ejecución, podría considerarse la *portabilidad* como un atributo de calidad interna del diseño, el que se cumplirá o no, pero no en una mejor o peor medida.

En cada proyecto se deben definir cuáles son los atributos de calidad a medir y cuáles de estos los de mayor importancia. Lo mismo sucede para el diseño.

El proceso de diseño tiene implícito un grupo de atributos que hacen más o menos atractivo y/o eficiente el diseño, tal y como se ha venido mencionando anteriormente. Estos atributos deben ser medidos empleando métricas de software (que serán vistas en momentos posteriores) y pueden especificarse, entre otros más, los que a continuación se muestran:

- Modularidad.
- Cohesión.
- Acoplamiento.
- Adaptabilidad del diseño a diferentes escenarios y bajo disímiles condiciones.
- Mantenibilidad (Facilidad de mantenimiento).
- Reutilización.
- Abstracción (Nivel de abstracción en el Diseño de Software).
- *Flexibilidad* del Diseño de Software.

1.4.2.1 MODULARIDAD

La modularidad es un concepto que ha venido ganando fuerza en la industria del software desde hace algunas décadas.

Se dice que le modularidad es el único atributo de software que permite gestionar un programa de forma intelectual (PRESSMAN, 2001), en tanto Booch la definía como *"la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados"* (BOOCH, 1998).

En cuanto al diseño se refiere, esta tiene que ver con la capacidad del mismo de separar en elementos (ya sean subsistemas, paquetes, componentes, etc.) tratados y nombrados por separados y que establecen algún mecanismo de integración para satisfacer los requerimientos del problema. Como mismo se dice que el software monolítico es difícil de construir por la cantidad de variables, la cantidad de rutas de control y la amplitud de referencias, además difícil de entender y por tanto difícil de mantener, el diseño monolítico le sucede exactamente lo mismo.

Se le debe sumar a todo esto que el esfuerzo por modularizar es directamente proporcional a la complejidad del problema, incluso es importante destacar, que para una complejidad *C* de un problema *p* que puede ser descompuesto en los sub-problemas p_1 y p_2 es mucho mayor si el problema se trata en una sola unidad a que si se analiza en dos partes diferentes (PRESSMAN, 2001). Esto es:

$$C_{(p_1+p_2)} > C_{(p_1)} + C_{(p_2)}$$

y por tanto el esfuerzo es:

$$E_{(p_1+p_2)} > E_{(p_1)} + E_{(p_2)}$$

Muy a pesar que este análisis pueda arrojar a la conclusión de que mientras más módulos mucho menor será el esfuerzo (coste), es importante aclarar que incide otro factor asociado a esto que da al traste con esta conclusión. Mientras más módulos (C_m), aumenta entonces el esfuerzo de integración de esos módulos (E_i). Luego, es necesario encontrar el punto crítico en el que el coste de esfuerzo por *modularizar* el diseño sea proporcional al coste de esfuerzo de integración

Sin embargo, la modularidad no se tiene en cuenta por una simple cuestión de desarrollar un diseño lo más modular posible, tratando de mantener los índices

de esfuerzos en un valor aceptable. Es preciso conocer entonces cómo se define un módulo con un tamaño adecuado y sobre la base de qué principios se puede construir. Bertrand Meyer (MEYER, 1997) define cinco criterios, cinco reglas y cinco principios para la definición de un módulo.

Criterios de Meyer para la Modularidad:

- **Descomposición (*Decomposability*):** La división de un problema en problemas más pequeños mediante un método de diseño, reduciría la complejidad de todo el problema y se llegaría a una solución modular factible.

- **Composición (*Composability*):** Un método de diseño satisface la composición modular si es capaz de ensamblar componentes reusables de otros sistemas, permitiendo así que se cumpla con uno de los principios de diseños de "no inventar nada que ya esté inventado".

- **Comprensibilidad (*Understandability*):** Es la propiedad que tiene un módulo de ser comprendido como una unidad autónoma o al menos lo suficientemente autónoma, como para ser construido y mantenido de manera relativamente sencilla.

- **Continuidad (*Continuity*.):** La continuidad modular es la capacidad que tiene un sistema de tener una repercusión localizada en un módulo y no generalizada en el sistema, ante cambios en los requisitos. Es decir, que el impacto de pequeños cambios en los requisitos, provocan pequeñas repercusiones en los módulos y por ende, una sencilla solución.

- **Protección (*Protection*):** Propone que las condiciones anormales ocurridas en un componente no se propaguen fuera del mismo, o a lo sumo a pocos otros. La protección está asociada al *ocultamiento de información*.

Reglas de Meyer para la Modularidad:

- **Correspondencia directa (*Direct Mapping*):** La estructura modular diseñada en el proceso de construcción del sistema, debe permanecer compatible con la estructura modular diseñada en la modelación del dominio del problema. Esto responde a dos de los anteriores criterios, a la *Continuidad Modular* y a la *Descomposición Modular*.

- **Pocas interfaces: (*Few Interfaces*)** La cantidad de relaciones entre módulos es la menor posible.

- **Interfaces pequeñas (bajo acoplamiento) (*Small Interfaces*):** La cantidad de elementos que forman la interfaz de un módulo es lo más pequeña posible, puesto que esos elementos actúan como puntos de dependencia.

- **Interfaces explícitas (*Explicit Interfaces*):** Las dependencias entre dos módulos son visibles, mirando a uno o el otro, o a ambos.

- **Ocultamiento de información (*Information Hiding*):** Los usuarios de un módulo necesitan conocer sólo una porción de la información empaquetada en el mismo.

Principios de Meyer para la Modularidad:

- **Principio de Unidades Modulares Lingüísticas (*Linguistic Modular Units*):** El lenguaje debe soportar la sintaxis para definir el concepto de módulo utilizado en el diseño. Por ejemplo, el concepto de *package (paquetes)* en Java.

- **Principio de auto documentación (*Self-Documentation*):** La documentación del módulo debería ser parte del módulo mismo (es preciso tener en cuenta que el módulo en sí mismo es información).

- **Principio de acceso uniforme (*Uniform Access*):** Todos los servicios ofrecidos por un módulo deben estar disponibles con una notación uniforme. No debe haber diferencia de sintaxis en las referencias a los miembros de una clase.

- **Principio abierto-cerrado (*The Open-Closed principle*):** Los elementos de software (clases, módulos, funciones, etc.) deben estar abiertos para extensiones y cerrados para modificaciones. Esto permite que se pueda extender el comportamiento cuando los requerimientos cambien y evitar que un cambio en un módulo provoque cambios en módulos dependientes (VÉLEZ SERRANO, y otros, 2005). Sin embargo, es preciso no absolutizar debido a que quizás existan elementos de éstos que no deban ser extendidos por cuestiones de seguridad para el sistema, eso deberá quedar a consideración del equipo de diseñadores.

- **Principio de única elección (*Single Choice*):**.En cualquier momento en que se realice una selección de alternativas, uno y solo un módulo del sistema debe conocer la lista exhaustiva de la las mismas.

Partiendo de lo anteriormente expuesto, se tendrían suficientes elementos para identificar hasta qué punto es conveniente definir los niveles de modularidad en un diseño, además de los elementos a tener en cuenta para construir un módulo de diseño.

1.4.2.2 COHESIÓN

La cohesión es una extensión natural del término *ocultamiento de la información.* En cuanto al diseño de software, se puede decir que un módulo es cohesivo, si ha sido diseñado (idealmente) para realizar una sola función dentro de un procedimiento de software y que requiere de poca interacción con otros procedimientos que se llevan a cabo dentro del mismo sistema pero en otros módulos (PRESSMAN, 2001).

La escala de medición del nivel de cohesión en un diseño no es lineal. Lo que quiere decir que un nivel *bajo* de cohesión no será mejor que un nivel *intermedio* y este a su vez, no mejor que un *alto* nivel, muy a pesar que el rango medio es tan "aceptable" como la parte más alta de la escala.

Lo real es que un diseñador de software no deberá preocuparse por la categorización de la cohesión en un módulo en específico, sino que deberá entender el concepto en sentido general del diseño.

Cuando los elementos de procesamiento de un módulo están estrechamente relacionados, y se ejecutan en un orden determinado, se está en presencia de una *cohesión procedimental*, así como también cuando estos elementos de procesamiento se centran en una estructura de datos en específico, la cohesión es de *comunicación*.

Como ya se había mencionado, lo más importante es intentar lograr una alta cohesión en el diseño e identificar cuándo se está en presencia de un nivel bajo de cohesión, para modificar el diseño y conseguir una mayor independencia funcional.

A esto último se le debe sumar, el hecho de la existencia de un patrón asociado a la cohesión (patrón *Alta Cohesión*) que permite la eliminación de los siguientes problemas:

- Dificultad de entendimiento.
- Dificultad de reutilización.
- Dificultad de mantenimiento.
- Afectaciones constantes por los cambios.
- Manejabilidad de la complejidad.

1.4.2.3 ACOPLAMIENTO

El acoplamiento es una medida en que se interconectan los módulos dentro de un sistema de software. El objetivo de éste es minimizar el grado de asociación entre diferentes abstracciones.

Constituye además una medida de la fuerza con que un elemento está conectado a otro, tiene conocimiento de otro elemento y *confía*[5] en otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de *demasiados*[6] otros elementos (IEEE, 2004).

En el diseño, se trata de conseguir un bajo acoplamiento entre los módulos, para lograr con esto un mejor entendimiento del sistema, evitar lo que Pressman llama *"efecto ola"* (PRESSMAN, 2001) refiriéndose a la propagación de errores y un diseño más fácil de mantener.

El concepto de acoplamiento viene asociado a otros conceptos como es la cohesión. En la práctica, el acoplamiento no puede considerarse de manera aislada a estos

[5] La confianza a la que se refiere, está asociada a la dependencia que existe en el empleo de funcionalidades definidas por otros elementos.
[6] En dependencia del contexto en que se esté trabajando. Es una definición relativa que da a la medida de una cantidad determinada.

conceptos, que además tipifican patrones de diseño como los son el *Experto* y el *Alta Cohesión.*

Existen un conjunto de formas en las que pueden manifestarse el acoplamiento en un diseño (LARMAN, 2003):

1. El tipo X tiene un atributo (miembro de datos, o variable de instancia) que hace referencia a una instancia de tipo Y, o al propio tipo Y.

2. Un objeto de tipo X invoca los servicios (métodos) de un objeto de tipo Y.

3. El tipo X tiene un método que referencia a una instancia de tipo Y, o al propio tipo Y, de algún modo. Esto, generalmente, comprende un parámetro o variable local de tipo Y, o que el objeto de retorno de un mensaje sea una instancia de tipo Y.

4. El tipo X es una subclase, directa o indirecta, del tipo Y.

5. El tipo Y es una interfaz y el tipo X implementa esa interfaz.

De manera general, el nivel de acoplamiento de un diseño determina la facilidad o no de mantenimiento al diseño, determina la facilidad o no de detectar errores en la implementación y cuán sencillo sea entender el diseño por los miembros del equipo de desarrollo y otros interesados.

1.4.2.4 ADAPTABILDIAD

La adaptabilidad del diseño está asociada a la capacidad del mismo de ajustarse a diferentes entornos bajo disímiles condiciones. Es un concepto muy asociado al nivel de flexibilidad del diseño (*epígrafe 1.4.1.8*), la facilidad de mantenimiento, el correcto uso de la abstracción y los niveles de acoplamiento y cohesión.

Un diseño fácilmente adaptable, reduce el tiempo de planificación, los costes de desarrollo, la usabilidad o reutilización del diseño, entre otros muchos aspectos.

1.4.2.5 FACILIDAD DE MANTENIMIENTO

La facilidad de mantenimiento es una cualidad que debe tener todo buen diseño, sin embargo esta no se obtiene *porque sí*, sino como consecuencia de un buen uso de mecanismos que favorezcan los conceptos del diseño como la *modularidad*, un correcto uso de la *abstracción*, una alta *cohesión* y un bajo *acoplamiento,* hacen del diseño un modelo cada vez más entendible y por tanto, fácil de mantener.

1.4.2.6 REUTILIZACIÓN

Existen varios criterios o atributos por los cuales se puede medir la efectividad o calidad de un determinado diseño. Según Roger S. Pressman:

> *"(...) el diseño de software orientado a objeto es difícil, y el diseño de software reusable es aún más difícil (...) el diseño debe ser específico al problema que se tiene entre manos, pero suficientemente general como para adaptarse a problemas y requerimientos futuros (...)"* (PRESSMAN, 2001).

Es evidente que lograr que un determinado diseño de software se ajuste a diferentes contextos y bajo diferentes condiciones es una labor de marcado esfuerzo. De manera que la reusabilidad (o reutilización, como en otros casos es conocido) es un elemento o característica que debe cumplir todo buen diseño.

En el contexto del desarrollo de software, cualquier elemento de software puede ser reutilizable, desde una clase en particular, un componente, *frameworks,* modelos, patrones, funcionalidades e incluso diseño. Se puede tipificar una reutilización en dependencia de lo que se esté reutilizando (ROSANIGO, 2000).

La reutilización de código es una de las más usadas y la menos metódica. Es lo que muchos conocen como el *"método"* de *cortar y pegar código fuente*.

La reutilización de componentes constituye una muy buena práctica debido a que el futuro del software depende en gran medida de la capacidad de reutilizar módulos, estructuras de datos, unidades funcionales o abstracciones específicas.

La reutilización en el diseño puede verse desde cualquiera de los niveles definidos por éste, ya sea desde un diseño de alto nivel (arquitectura), como la reutilización de algún *framework*; a un nivel de objetos y clases; a nivel de componentes o desde la propia interfaz de usuario.

Es importante señalar que el grado de reutilización en un diseño no debería verse como una cota a alcanzar, debido a que la reutilización potencial parece variar de manera brusca con el dominio y está afectada por muchos valores no técnicos, como presiones de la planificación, naturalezas de las relaciones con los subarrendatarios y consideraciones de seguridad (BOOCH, 1998). Sin embargo, esto no quiere decir que se deje de aspirar a un alto grado de reutilización en el diseño, lo que no se debe hacer es establecer una cota numérica para el alcance de dicha reutilización.

Grady Booch plantea la institucionalización de la reutilización como vía para desarrollar dicho concepto, partiendo de que el mismo no surge *porque sí* sino que las oportunidades para reutilizar deben buscarse y recompensarse (BOOCH, 1998).

Anteriormente se hablaba sobre el enfoque que había que darle a la calidad del proceso de diseño y el empleo del tiempo y coste de desarrollo. Retomando aquello, se decía que el proceso debiera estar enfocado a la calidad para alcanzar buenos niveles en cuanto a tiempo y costes de desarrollo. A esto es preciso agregarle que si definen estrategias para lograr que el diseño sea lo más reutilizable posible –lo que se traduce, entre otras cosas, en el buen uso de patrones de diseño-, la calidad del producto (tanto del software como del diseño) se verá incrementada y el tiempo y los costes se verán reducidos gracias a la reutilización.

Haciendo una valoración en el marco de la facultad 3 de la Universidad de las Ciencias Informáticas, es válido decir que uno de los principales problemas que afecta la dinámica del desarrollo de software reflejado en el tiempo de desarrollo de los sistemas, en la estandarización del empleo de componentes predefinidos y por tanto en la calidad del producto de software final, es el tema de la reutilización. Y esto tiene mucho que ver con la no institucionalización de este concepto, impidiendo la existencia en algunos casos, de repositorios de componentes reutilizables, tanto para el diseño, como para la implementación e incluso hasta la arquitectura. Sobre esto han venido realizándose algunos valiosos esfuerzos como muestra de la intención de dar solución factible a esta problemática.

De manera general, la reutilización cuesta recursos a corto plazo pero los ahorra a largo plazo. Ésta solo tendrá éxito en organizaciones que adopten una visión a largo plazo del desarrollo de software y optimicen los recursos para "algo más" que para el proyecto actual.

La reutilización tiene un impacto positivo en el proceso de desarrollo de software por todo lo anteriormente expuesto, agregando además los beneficios alcanzados mediante ésta, en términos de planificación.

1.4.2.7 NIVEL DE ABSTRACCIÓN

La *abstracción* es un mecanismo muy asociado al de *generalización* y que permite la descripción de un problema empleando conceptos que son familiares en el entorno de dicho problema y sin tener en consideración detalles irrelevantes de bajo nivel.

La IEEE define la abstracción como *"el proceso de ocultamiento de la información que permite que elementos diferentes sean tratados de la misma manera".* En tanto contextualizaba este mismo concepto en el diseño de software definiéndole dos mecanismo de abstracción: la *parametrización* y la *especificación*. En el caso de la

última categoría de abstracción, identificaba tres niveles de abstracción para el proceso de diseño de software (IEEE, 2004):

Niveles de abstracción por especificación:

- Nivel de abstracción de datos.
- Nivel de abstracción de procedimiento.
- Nivel de abstracción de control.

A medida que se va avanzando en el proceso en cuestión, se van entrando cada vez más en un nivel más bajo de abstracción, hasta llegar a la implementación del sistema.

La abstracción puede ser vista desde dos niveles de generalización, partiendo de un análisis del proceso de diseño propiamente dicho. Se podría definir un nivel de abstracción asociado al *macro diseño* –diseño de alto nivel (arquitectura) – y un nivel de abstracción asociado al *micro diseño*.

A raíz de la existencia de diferentes niveles de abstracción, es válido aclarar que no debieran mezclarse elementos de diferentes niveles de abstracción debido a que podría generar confusión en el diseño.

Para el nivel de abstracción de datos, se definen colecciones de datos nombradas que describen un *objeto* en específico. Ejemplo de la abstracción *automóvil* que define un concepto en el que podría venir acompañado de atributos como lo pueden ser el *color* de automóvil, la *marca, año* de fabricación entre otros.

Para el caso del nivel de abstracción procedimental y una vez visto el nivel de datos, se puede asumir que el primero es prácticamente análogo debido a que define una colección de *procedimientos* nombrados que realizan una función en específico y que además hacen uso de la información contenida en los atributos del nivel de abstracción de datos. Éste se encuentra en un nivel inmediato inferior de la abstracción de datos.

Por último, el *nivel de abstracción de control* implica un mecanismo de control de programa sin especificar los datos internos.

1.4.2.8 FLEXIBILIDAD

La flexibilidad es un término que comparte cierta analogía con la facilidad de mantenimiento y la adaptabilidad, y como éstos, es una consecuencia de un correcto uso de conceptos como el acoplamiento, la cohesión, la modularidad y la abstracción, además de un correcto empleo de mecanismos como los patrones de diseño.

Está asociada a la capacidad del diseño de cambiar sin provocar grandes *"traumas"* o transformaciones en el resto del modelado.

La flexibilidad tributa a la facilidad de mantenimiento en cierta medida, sin embargo respecto a esto tiene solo un inconveniente, a mayor flexibilidad menor es la entendibilidad y la analizabilidad del diseño, por lo que para este caso el coste de mantenibilidad es un poco mayor.

Evidentemente mientras más simple es el diseño más entendible se torna el análisis. La mantenibilidad parte de igual manera de un principio de simplicidad del diseño, sin embargo, existen otros aspectos a tener en cuenta para lograr desarrollarla de manera efectiva.

La flexibilidad en el diseño facilita la mantenibilidad en el sentido de la no propagación de cambios, pero como es lógico aumenta la complejidad del diseño, los costes de implementación y los costes de mantenimiento en dependencia de la complejidad del cambio, pero estos últimos aspectos son un efecto normal de introducir más flexibilidad en un diseño a cualquier nivel.

1.4.3. EMPLEO DE PATRONES EN EL PROCESO DE DISEÑO DE SOFTWARE

Utilizar algún patrón de diseño en la construcción de un software sin analizar su efectividad y consecuencias, no garantiza que se obtenga un producto con una alta calidad. No es usar por usar patrones de diseño, sino usar aquellos que sean los más adecuados. De manera que es importante señalar que los patrones que vayan a utilizarse en la definición del diseño del sistema, deberán ser analizados para evitar incorrectas selecciones.

Debido a la misma signatura con que se define un patrón, es posible conocer si éste puede o no responder a la solución de determinado problema en específico ya que el patrón mismo lo describe, por lo que se puede inferir, que nunca se debería seleccionar un patrón con el fin de resolver un problema para al cual no ha sido creado. Además de esto, se impone verificar que las consecuencias para el uso de determinado patrón no constituyen una afectación significativa en la construcción del sistema. O sea, que el impacto negativo que cause el patrón aplicado a determinado atributo de calidad no sea significativo o no conlleve a la no correspondencia con los requerimientos del sistema en construcción.

Una vez seleccionado un patrón, en caso que se desee hacerle modificaciones al mismo, pues ya se estaría en presencia de un nuevo patrón de diseño, sin embargo, éste puede resolver el mismo problema u otro que sea definido, en este último caso, se estaría en presencia de un patrón oxímoron (LARMAN, 2003).

Cuando se selecciona un patrón para dar solución a un problema determinado y éste no está acorde con esa determinada situación, se estará en presencia de un grave riesgo reflejado en la posibilidad real de que el diseño no sea lo suficientemente modular, o lo suficientemente flexible, o presente un alto grado de acoplamiento o un bajo nivel de cohesión.

Los patrones no constituyen una teoría y mucho menos un lenguaje de programación, sencillamente son la experiencia obtenida en el proceso de desarrollo de software, probada y que funciona realmente.

Usar patrones de diseño no es una acción trivial dentro de todo el proceso de desarrollo de software. Se requiere de un profundo análisis para poder identificar el patrón correcto.

Los patrones de diseño tienen una fuerte relación con los atributos de calidad. De hecho, puede afirmarse que estos en muchos casos son tipificados por los propios atributos de calidad del diseño.

Algunos autores señalan que los patrones de diseño tienden a operar independientemente de un paradigma en particular o de un lenguaje de programación. En realidad esto no es del todo correcto debido a que los patrones precisamente responden a un paradigma en particular y en muchos casos, se ajustan más a un determinado lenguaje de programación que a otro. Un ejemplo concreto de esto, son los *Core J2EE Patterns* publicados por la *Sun Microsystem* en el 2001, los que contienen las mejores soluciones para ayudar a los desarrolladores de aplicaciones empresariales a diseñar y construir aplicaciones sobre la plataforma J2EE.

Sobre el empleo de patrones existe otro elemento de medular importancia: el coste. Evidentemente el empleo de patrones imprime un mayor nivel de flexibilidad al diseño. Esto puede verse reflejado en la introducción de clases de servicios en un diagrama de clases que resuelve determinado problema, que permiten hacer de este diseño un modelo aún más extensible y flexible. Mediante empleo de patrones, podrán agregársele muchos más clases de servicios o auxiliares –como se le desee llamar- al modelo, incluso para un mismo número de clases. Por ejemplo, para la modelación de un proceso de venta de seguros empresariales, podría estarse pensando en dos clases: *vendedor* y *seguro empresarial.* ¿Qué sucedería si en un futuro se quisiera agregar ventas de otros tipos de seguros?, e incluso mejor aún,

¿qué sucedería si en un futuro se desease vende hasta otro tipo de producto? Evidentemente a este diseño para estas dos clases, se le debería incorporar otros mecanismos que hagan posible extender la funcionalidad del mismo.

Partiendo de lo anteriormente planteado, se puede llegar a la conclusión de que aplicar patrones podría sacrificar en cierta medida la *analizabilidad* del diseño. Es además significativo señalar que el coste de implementación inicial aumenta según se complica el diseño. Añadir puntos de flexibilidad cuesta. Si un diseño es muy flexible –normalmente esto se logra aplicando patrones- tendrá una gran capacidad de adaptación y será más estable, pero se perderá en entendibilidad, lo cual será absolutamente opuesto si no se aplican patrones (CABRERO, y otros, 2008).

Hasta aquí se ha analizado en sentido general, algunas consideraciones a tener en cuenta para el empleo de patrones en el proceso de diseño de software. Más adelante se estarán viendo ejemplos de aplicaciones de algunos patrones de diseño.

1.5 MÉTRICAS DE DISEÑO DE SOFTWARE PARA LA TOMA DE DECISIONES

Las métricas constituyen un mecanismo esencial para la toma de decisiones dentro de alguna organización, en función del mejoramiento de los procesos que se llevan a cabo dentro de ésta, asociado a estimación de tiempos, planificación y demás; o dirigido al mejoramiento de la calidad de los productos que se obtienen dentro de dicha organización.

El término métrica suele confundirse con medición o medida, aunque tienen estrecha relación, son tres conceptos diferentes.

La medición "*es el proceso por el cual los números o símbolos son asignados a atributos o entidades en el mundo real tal como son descritos de acuerdo a reglas claramente definidas*" (FENTON, y otros, 1997). En tanto una medida "*proporciona*

una indicación cuantitativa de extensión, cantidad, dimensiones, capacidad y tamaño de algunos atributos de un proceso o producto" (PRESSMAN, 2001).

El estándar de la IEEE *610.12-1990 (IEEE Standard Glossary of Software Engineering Terminology-Description)* define el concepto de métrica como "*una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado*" (IEEE, 1990).

En este epígrafe, se estará haciendo un análisis de la evaluación de la calidad desde sus dos enfoques: orientada al producto y orientada al proceso.

1.5.1 MÉTRICAS DE DISEÑO ORIENTADA A LA CALIDAD DEL PRODUCTO

Desde hace más de una década se han propuesto diferentes métricas orientadas a objetos (OO), pretendiendo, en muchos casos, medir el diseño más que el código u otros productos del ciclo de vida del desarrollo de software.

En este apartado se analizarán solo algunas de las métricas de diseño orientadas a la calidad y no a la productividad. Es decir, aquellas que de de manera cuantitativa, dan la medida del nivel alcanzado de ciertas propiedades supuestamente deseables del diseño orientado a objetos (DOO), tales como encapsulamiento, abstracción, cohesión, acoplamiento, entre otras más. En este caso, no será preciso entrar en detalles sobre las métricas, de hecho, no es un objetivo de este epígrafe, mas si sería conveniente mencionar algunos aspectos de relevante significancia, como las características de estas métricas, su finalidad, la relación que guardan con características y sub-características de calidad definidas por el modelo de calidad ISO 9126 (*ver ANEXO 1*), entre otros aspectos.

Es importante señalar, que con la aparición de conceptos como los patrones de diseño y la refactorización, surgieron nuevos elementos para juzgar cuándo se está en presencia de un buen o mal diseño.

Las métricas, naturalmente, responde a algún atributo de calidad en específico, de hecho, éstas precisamente son tipificadas por los propios atributos de calidad que miden. El resultado de evaluar una métrica de diseño sirve para la toma de decisiones en función de éste proceso.

La familia de métricas de *Shyam R. Chidamber* y *Chris F. Kemerer* (*ver ANEXO 2*) fue el primer intento de definir mecanismos para evaluar la calidad de software orientado a objetos exclusivamente. Los que se centraban en medir el acoplamiento, la cohesión, complejidad y comunicación entre clases (R. CHIDAMBER, y otros, 1994). Estas métricas ayudan a asegurar la mantenibilidad de los productos de software, ya que dependiendo de esas medidas se puede predecir si el diseño y su posterior implementación serían fáciles de corregir, mejorar o adaptar a nuevos requisitos (MARÍN, y otros, 2007).

Estudios posteriores, como el de *Li* y *Henry* (*ver ANEXO 3*), propusieron otros modelos más ajustados al mejoramiento de la mantenibilidad del software, por una parte incluyéndole al anterior modelo citado los atributos de *abstracción* y el *tamaño del diseño*, y por otro lado, realizando modificaciones a la métrica *Lack of cohesion in methods (LOCM)* de *S. R. Chidamber* y *C. F. Kemerer* para medir el nivel de cohesión entre métodos de clases, ya que consideraron tenía un carácter ambiguo (OLMEDILLA ARREGUI, 2005).

Sin embargo, ninguno de los dos modelos anteriores es completamente orientado al diseño. No es hasta que *Bansiya* y *Davis* presentan el modelo para la evaluación de atributos de calidad del diseño denominado *Quality Model for Object-Oriented Design (QMOOD)*. Este modelo está divido en cuatro niveles. Un primero nivel (L1) para el análisis de los atributos de calidad del diseño; un segundo nivel (L2) referente a las propiedades del diseño; el tercer nivel (L3) asociado a las métricas de diseño y por último, un cuarto nivel (L4) asociados a los componentes de diseño. Además, este modelo define relaciones entre componentes de niveles con el objetivo de facilitar su aplicación (JETTER, 2006).

Existen otros muchos modelos de calidad para el diseño, incluso hasta herramientas que permiten la automatización de la evaluación de la calidad del diseño, como es el caso del modelo RARER de *Barber* y *Grase,* el que se encarga de crear un modelo de evaluación de la calidad mediante la especificación del usuario de los atributos de calidad a medir.

Se podrían citar otras métricas de calidad orientadas al diseño, como las métricas de *Lorenz* y *Kidd* (*ver ANEXO 4*), las que tenían como objetivo medir las características estáticas del diseño. O las métricas de *Merchesi* (*ver ANEXO 5*) para medir la complejidad del sistema mediante el balanceo de responsabilidades entre paquetes y clases, y la cohesión y el acoplamiento entre entidades del sistema. Otra métrica orientada al acoplamiento era la de *Harrison* (MARÍN, y otros, 2007). Todas las anteriores métricas responden a la *mantenibilidad* del diseño.

De a cuerdo al previo análisis realizado sobre algunas de las métricas para el diseño, puede decirse que una gran parte de éstas necesitan del código fuente para su evaluación, sin embargo, no se podría absolutizar debido a que existen un gran número de métricas de diseño, como algunas de las antes mencionadas, que pueden ser evaluadas solo sobre algún diagrama de clases UML, como es la métrica de *Genero* (*ver ANEXO 6*), para analizar la complejidad del diagrama UML de clases partiendo de los diferentes tipo de relaciones entre clases.

Sin embargo, hablar de diseño, no solo enmarca la definición de diagramas estáticos como los de clases, sino que aspectos dinámicos como los diagramas de estados y de secuencias se encuentran incluidos dentro del proceso. Existen también métricas orientas a la evaluación de estos diagramas, como las métricas de *JA Cruz-Lemus (ver ANEXO 7)* para garantizar la *comprensibilidad* de los modelos conceptuales y las métricas de *Kiewkanya (ver ANEXO 8)* para mediar las interacción entre clases, respectivamente.

1.5.2 MÉTRICAS DE DISEÑO ORIENTADA A LA CALIDAD DEL PROCESO

Existen cuatro responsabilidades o etapas claves a la hora de evaluar un proceso: definir, medir, controlar y mejorar el proceso, tal y como se muestra en la *fig. 1.5.2.1* (MONASOR, 2008).

La gestión de los procesos es un factor clave para incrementar la productividad del equipo de desarrollo y la calidad del producto. Por esta razón, puede afirmarse que la calidad de los procesos incide directamente en la calidad del producto general.

De la misma manera que existen modelos de calidad orientados al producto y más específicamente al diseño. Por otro lado aparecen los modelos de calidad orientados al proceso así como también las métricas orientadas al mismo objetivo.

Norman E. Fenton hace una propuesta interesante de las clasificaciones de atributos de calidad orientados al control de la calidad del producto, del proceso e incluso de los recursos (FENTON, y otros, 2005).

Se podrían citar diferentes modelos orientados a evaluar la calidad de los procesos. Estos modelos de calidad definen principios, etapas y mecanismos que pueden ser ajustados al proceso de diseño de software propiamente dicho.

Muchos autores dan un enfoque dual al proceso de calidad de software, lo que naturalmente es lógico dado a que la calidad final del producto depende en gran medida de la calidad del proceso mediante el cual se desarrolla.

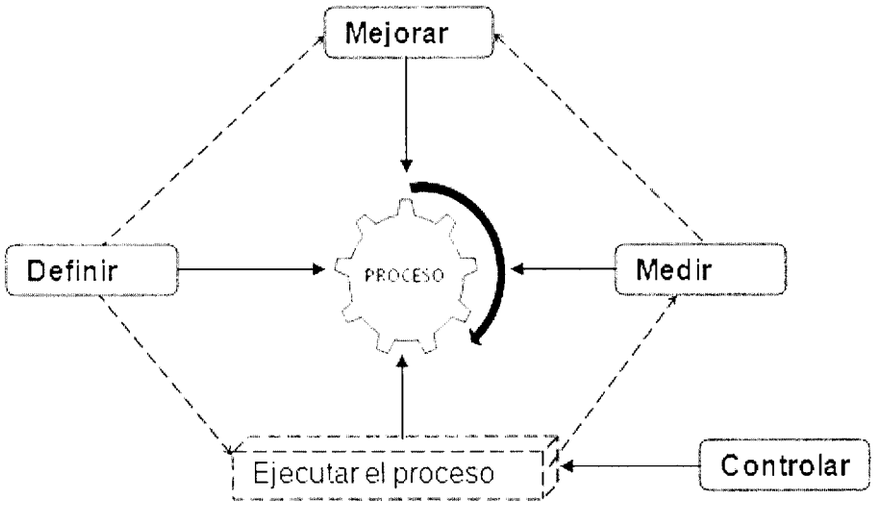

Fig. 1.5.2.1 Etapas para evaluar un proceso.

La IEEE 1061 presenta una metodología que sirve no solo para evaluar productos de software, sino que también incluye métricas para analizar y validar la calidad de procesos de software (MONASOR, 2008).

Otro importante modelo es CMM, que presenta diferentes niveles que deberán cumplirse para alcanzar determinado grado de madurez. Sin embargo, este modelo tiene un inconveniente, él indica qué hacer, pero no cómo hacerlo, además de que especifica la necesidad existente de métricas pero no aporta ninguna en específico.

PSP (Proceso de Software Persona) es otro de los modelos que se basa en la mejora de los procesos centrándose específicamente en el ingeniero de software y a través del mejoramiento de las actividades y la productividad de éstos.

CONCLUSIONES

Hasta este entonces, se ha realizado un estudio sobre lo que es el diseño de software orientado a objetos, su evolución y la manera en que es concebido por diferentes empresas, metodologías de desarrollo de software e incluso otros autores de autorizado criterio en el mundo de la industria de sistemas informáticos.

Partiendo de todo esto, se definió en el presente trabajo una contextualización del proceso de diseño, debido a la carencia en otros conceptos sobre la especificación de dónde se enmarca dicho proceso. De la misma manera en que se definen un conjunto de perspectivas de diseño, con el objetivo de facilitar el análisis y desarrollo de esta importante actividad dentro del ciclo de vida.

Por último, se adentró en el tema del análisis de los atributos de calidad del diseño y las métricas orientadas a la evaluación de dichos atributos. En este caso, se llegaron a conclusiones importantes ya antes mencionadas. Primero, que la mayoría de estas métricas, e incluso, la mayoría de los modelos de calidad orientados supuestamente al diseño, necesitan de la participación de otras etapas del ciclo de vida como lo es la implementación, dado a que requieren del código fuente para realizar los cálculos. Lo que hace pensar que no son completamente o más bien exclusivamente orientadas al diseño. Aunque existen modelos, como el de *Bansiya* y *Davis* que si son absolutamente orientados a este proceso.

Otra importante conclusión sobre las métricas es que la mantenibilidad es el indicador de calidad más utilizado, lo cual es lógico dado que la orientación a objetos es un paradigma que estimula la *flexibilidad* y la *reutilización*. Otros indicadores definidos por la ISO 9126 como la *eficiencia* y la *portabilidad* no se tienen en cuenta, lo que puede deberse a que estos parámetros no guardan mucha relación con el desarrollo del diseño.

De manera general, estos son algunos aspectos conclusivos para el presente capítulo recién culminado.

CAPÍTULO 2

Capítulo 2: EL DISEÑO DESDE UN ENFOQUE INDUSTRIAL

INTRODUCCIÓN

Hasta este momento se han establecidos las bases teóricas y los diferentes análisis sobre el proceso de diseño de software orientado a objetos (DOO). Cuestiones como la conceptualización y la contextualización del mismo permiten centrar la atención en aspectos más específicos desde un enfoque industrial y orientado a dar solución al problema que anima la dinámica del presente trabajo de diploma.

En un primer momento se estará haciendo referencia especialmente a las especificidades de las dimensiones o perspectivas del proceso de diseño mencionadas en el capítulo anterior, así como los criterios que deberán ser tenidos en cuenta, partiendo de los propios atributos de calidad del proceso, a la hora de analizar dichas dimensiones y esencialmente desde un punto de vista del proceso insertado en la industria del software.

Se retomarán elementos entre los que se encuentran los conceptos de modularidad, reutilización, robustez, mantenibilidad y métricas de calidad, además de otros aspectos como son la composición y arquitectura, relacionada esencialmente con la modularidad del diseño desde un ámbito estilístico así como la relación existente entre el proceso en cuestión y las diferentes taxonomías arquitectónicas, sobre todo con los estilos de codificación y empaquetamiento.

Otro elemento de marcada importancia es el de las técnicas de diseño, cómo debe organizarse un equipo de diseño y la definición de mecanismos de interacción, qué herramientas utilizar, cómo discutir, cómo establecer y controlar estándares y todo

esto desde una perspectiva *hombre* y *equipo*, cuestión que le imprime un atractivo indiscutible al presente capítulo.

Por último, pero no de restada importancia, es la propuesta de la estructura organizacional de un equipo de diseño definida desde un enfoque en *departamentos,* los que básicamente representan los principales sub-procesos de diseño a realizar y modelada su interacción usando BPMN, además de la propuesta de roles de trabajo con sus respectivas actividades y artefactos correspondientes. Sobre este último aspecto, la definición del Expediente de Diseño como elemento que reagrupará los principales artefactos del proceso.

2.1 LAS DIMENSIONES DEL DISEÑO DE SOFTWARE

Observar el proceso de diseño en dimensiones, o lo que es lo mismo, en perspectivas, permite ampliar el abanico temático sobre los elementos a tener en cuenta para desarrollar un buen diseño. Es decir, es preciso dejar de ver al diseño solo como la *simple actividad* de modelar lo que se pretender implementar y comenzar a verlo como lo que es realmente, un *proceso* con todo y el significado semántico de la palabra.

Sobre el proceso de diseño inciden aspectos que tipifican estas dimensiones anteriormente definidas –marco histórico, marco tecnológico, interrelación de elementos de software, seguridad y rendimiento e interfaz- tales como los paradigmas y lenguajes más usados en el momento en que se enmarca el proceso, la selección de *frameworks* de trabajo, la automatización del código, la selección y uso de patrones, entre otros.

A medida que los sistemas de información son más avanzados, los clientes requieren mayor variedad de servicios lo que conduce a su vez a mayores y más complejos sistemas. Para poder responder a estas demandas con la rapidez que el momento amerita, el software debe construirse de acuerdo a esquemas de representación y diseños que proporcionen una potente capacidad expresiva y resulten fáciles de comprender, extender y reutilizar. De ahí la necesidad de pensar en el diseño como un proceso industrializado, en el que los diseñadores –al igual que los desarrolladores en sentido general- fungirían como "obreros" de una fábrica, pero en este caso de software, produciendo a partir de modelos, estándares y normas. Sin embargo, esto nunca deberá limitar la capacidad creativa de los equipos de desarrollo. Producir software a escala industrial, no es como producir autos, no es montar una línea de producción de ensamblaje secuencial y mucho menos es *ejecutar sin pensar* solo por el hecho de que está normado. Es sumamente importante, que esta industrialización conciba flexibilidad en su definición.

Industrializar el proceso de diseño, primero que todo abarata los costes y reduce el ciclo de desarrollo, y en segundo lugar, además de ser una consecuencia de esto mismo, es un enfoque de producción que responde a las demandas de un mercado tan volátil como es el de software y se apoya enormemente en unas de las bondades del diseño orientado a objeto, la reutilización.

Para que se entienda con claridad el proceso de industrialización del diseño, es preciso comprenderlo desde un enfoque en perspectivas de diseño.

2.1.1 MARCO HISTÓRICO

Evidentemente el momento histórico en que se lleva a cabo algún proceso de desarrollo ejerce una influencia considerable e incluso definitoria sobre dicho proceso. Sin el menor ánimo de filosofar, es preciso se reconozca **el *momento* como parte indisoluble del *proceso***, amén de que el primero brinda conceptos, herramientas y marcos de trabajo para llevar a cabo la dinámica de este último.

La industria del software no es un "*fenómeno*" ajeno a la economía de mercado, las nuevas concepciones de organización del trabajo, la legislación de tarifas de impuestos, la situación de empleo y/o desempleo, entre otros factores; sino todo lo contrario, se encuentra determinada por ellos.

Desde esta perspectiva, es preciso desarrollar estudios de mercados para evaluar las principales tendencias sobre la concepción y organización del trabajo, la definición de criterios para la selección de la tecnología a emplear basándose en los requerimientos de los clientes, los paradigmas y lenguajes de desarrollos más usados; así como también un estudio sobre la automatización del proceso de diseño, muy ligada a la selección de la tecnología y en relación demás con la generación de código fuente.

En este punto, es importante hacer referencia al modo y organización del trabajo, la manera de concebir una estructura de organización y las estrategias a seguir para responder a las demandas de mercado. En epígrafes posteriores se estará tratando este tema con más énfasis y detalles.

Finalmente, la búsqueda y definición de algún enfoque de trabajo es un tema primordial. En este caso se pueden encontrar dos de los enfoques más usados dentro de la propia industria del software, por un lado el enfoque MDA (*Model Driven Architecture*) propuesto por el consorcio OMG (*Object Management Group*) y por otro lado las Factorías de Software, definidas y promovidas por la Microsoft (MUÑOZ, y otros, 2007).

Ambas propuestas tienen sus características distintivas que la enfocan a dos situaciones diferentes, por una parte MDA es adecuado cuando la interoperabilidad con otras herramientas o el uso de herramientas existentes (que sigan los estándares de OMG) sea un factor clave y por otro, las Factorías de Software se recomiendan cuando existe la intención de construir una serie de sistemas similares y/o se va trabajar dentro de un dominio determinado; éstas tiene un gran atractivo debido a que son capaces de construir una programación basada en un modelo sugerido, con una alta calidad y en muy poco tiempo, reproduciendo en cierta forma los esquemas de fabricación industrial basados en la normalización, la reutilización de componentes, el diseño basado en patrones y la homologación de los procesos de fabricación (I-Sol S.A. Intelligent Solutions, 2008)

Ahora bien, qué sucede cuándo se está en presencia de ambas situaciones. Análisis relacionados con este último aspecto, se estará tratando de manera más particular en epígrafes posteriores.

En esencia, el marco histórico no es más que el contexto en que se desarrolla el proceso de diseño, de manera que se pueda realizar un estudio sobre los aspectos de mercado que inciden sobre la dinámica del proceso; paradigmas y lenguajes a emplear en el mismo, incluyendo el empleo de patrones de diseño; métodos de

organización del trabajo, en relación con enfoques de producción; y criterios de selección de la tecnología a emplear para la automatización del proceso de diseño y generación de código.

2.1.2 MARCO TECNOLÓGICO

La complementariedad es una característica de este enfoque en perspectivas o dimensiones del diseño. Esto hace que los aspectos tecnológicos del marco histórico sean calzados con más detalles en esta otra perspectiva.

El diseño detallado parte de una arquitectura previamente definida. Esta última envuelve un conjunto de decisiones estratégicas de diseño, lineamientos, reglas y patrones que restringen el diseño y la implementación de un software.

Existen autores que tratan la arquitectura como un proceso desligado del diseño cuando en realidad es una manera de expresión del diseño, o lo que es lo mismo, la representación del macro diseño o el diseño de alto nivel. La arquitectura forma parte del diseño y no viceversa, de hecho, se ve representada en una de las etapas del modelo de diseño como la *etapa de diseño arquitectónico* (PRESSMAN, 2001).

Una vez que es definida la tecnología a emplear en la construcción del sistema es necesario realizar un análisis sobre la incidencia de ésta en la construcción de un modelo de diseño detallado. Este análisis debe estar esencialmente dirigido al empleo de patrones de diseño, partiendo del principio de que muchos patrones no pueden ser aplicados a determinado lenguaje de programación, provocando que existan patrones específicos de un lenguaje o de alguna tecnología de desarrollo, incluso hasta de un paradigma en específico, como es el caso de los patrones de diseño para PHP, J2EE o los de la Programación Orientada a Aspectos (*AOP, Aspect Oriented Programming*) respectivamente.

En resumen, el marco tecnológico del diseño brinda un entorno de análisis de dos aspectos esenciales relacionados con la tecnología, bien sea de lenguajes, *frameworks* y paradigmas a emplear y el impacto en el diseño, como de la propia tecnología encargada de automatizar el proceso de diseño y la generación de código fuente.

2.1.3 INTERRELACIÓN ENTRE ELEMENTOS DE SOFTWARE

Esta dimensión está dirigida especialmente al diseño a niveles de datos, arquitectura y componente.

Luego que se transforma el modelo de dominio de información que se crea durante una etapa previa de análisis, en las estructuras de datos que se necesitarán para desarrollar el sistema, éstas son organizadas e interrelacionadas conformando un diseño estructural de la aplicación listo para adentrarse en otra etapa de especificación a un nivel de componente.

Desde esta perspectiva se tienen en cuenta tres elementos fundamentales, primero la definición de los patrones de diseño que se pueden usar para lograr los requisitos del sistema y en segundo lugar, las restricciones que afectan la manera en que se puedan aplicar estos patrones. Ya para este momento estas restricciones no estarían relacionadas con la tecnología que se emplearía para desarrollar el sistema puesto a que este análisis se realizó en la perspectiva de *marco tecnológico*, sino que más bien se enfocaría el análisis a los requerimientos que deberá tener el sistema, es decir, a la conveniencia de aplicar uno u otro patrón para alcanzar determinado nivel de acoplamiento, cohesión, modularidad, etc.

Sin embargo, en el contexto de este enfoque deberá prestarse marcada atención a un tercer tema cuyo objetivo sustenta la dinámica de esta perspectiva: el *tratamiento tecnológico a las responsabilidades identificadas* en el sistema. De este tema se

derivan dos términos significativos, por un lado *responsabilidad*[7] y por otro *tratamiento*[8].

Sobre la *responsabilidad* Larman decía que éstas contienen uno o más propósitos u obligaciones de un elemento de software (LARMAN, 2004), como por ejemplo, la responsabilidad de asegurar que no puedan existir en el sistema dos instancias en ejecución de una misma clase, lo que conlleva a la aplicación de algún tratamiento tecnológico que permita que esta responsabilidad sea completamente satisfecha. Como es el caso de crear claves primarias para cada instancia en el esquema de bases de datos que permitan controlar la creación de éstas de una determinada clase o simplemente definir un atributo de clase como mecanismo para controlar la cantidad de instancias creadas en el sistema.

El diseño basado en aplicar tratamientos a responsabilidades fue propuesto a finales de la década del 1980 y principios de los 90 y acogido preferentemente por el paradigma orientado a objetos (OO).

Respecto a las responsabilidades, pueden clasificarse en dos grandes grupos partiendo del modelo de diseño:

- *Responsabilidades explícitas*: son aquellas que pueden observarse a partir del propio modelo de diseño, ya sea de los diagramas de clases, de estados, secuencia o interacción, entre otros. Ejemplo la cardinalidad entre dos clases, la herencia, definición de constantes, etc.

- *Responsabilidades implícitas:* son las que no aparecen en el modelo de diseño pero pueden ser especificadas en un lenguaje natural y podrían dividirse en:

[7] Entiéndase como responsabilidad de un elemento de software.
[8] Refiérase a tratamiento tecnológico.

- *Invariantes:* aquellas que deban cumplirse en cualquier estado válido del sistema.
- *Eventuales:* asociadas a la definición de alguna funcionalidad o método del sistema y pueden ser a *priori* o a *posteriori.*

Esta dimensión del diseño es una de las más importantes, a ella se asocian un conjunto de artefactos generados en el proceso y que recogen los tres elementos fundamentales de esta perspectiva: análisis de los patrones de diseño que pueden ser aplicados; selección de aquellos que respondan al cumplimiento de los requerimientos del sistema y la aplicación de tratamientos tecnológicos a responsabilidades definidas en el mismo.

2.1.4 RENDIMIENTO Y SEGURIDAD

En esta dimensión se centrará el análisis en los temas de programación segura y rendimiento del sistema, tal y como lo sugiere el propio nombre de la perspectiva.

El tratamiento del tipiado de datos, el uso de elementos externos de la aplicación (como ficheros XML, accesos a bases de datos para *cargar* alguna información, etc.), el empleo de las diferentes estructuras de datos, entre otros elementos, deberán tenerse en cuenta para lograr que el futuro software funcione de manera eficiente y de acuerdo a los tiempos de respuestas mínimos necesarios.

La seguridad es otro aspecto a tener en cuenta, sin embargo, no se refiere a la seguridad sobre los permisos o roles de acceso al sistema sino más bien a la programación segura. Este es un tema relacionado con el *diseño procedimental,* ya que para este entonces se estaría en un nivel de abstracción (de detalle) del diseño muy próximo a la implementación del software en cuestión.

La programación segura podría dividirse en dos etapas, una primera etapa correspondiente a un nivel de abstracción más alto y asociado más específicamente

al diseño y una segunda etapa en el momento en que se comienza a programar el sistema. Por lo que se estaría hablando del *diseño seguro* e *implementación segura*.

Evidentemente la implementación del sistema responde al diseño que se realice de éste. De la misma manera, los elementos de seguridad (como la validación de operaciones, tratamiento de errores, control de la finitud de los algoritmos, etc.) que deberán tenerse en cuenta a la hora de rescribir el código, también responden al diseño previo aunque el que se satisfagan todos los requerimientos depende en gran medida de situaciones objetivas como lo es la responsabilidad de los programadores. Debido a esto último, es preciso desarrollar algún mecanismo de gestión y control de errores del código (como pruebas de conceptos y pruebas unitarias) que permita garantizar dicha seguridad.

Si bien las metodologías y estándares de desarrollo de software buscan tener altos niveles de confiabilidad y control de la solución informática, los elementos de seguridad y principios de diseño seguro no constituyen parte formal de dichos estándares y metodologías. En este sentido el diseño podría verse comprometido desde un punto de vista de seguridad.

Es razonable que la seguridad orientada al desarrollo de software responda a los principios elementales de seguridad de la información. Siguiendo este criterio, se proponen un grupo de cuatro buenas prácticas de diseño enfocadas en la *seguridad*:

- *Principio de la asignación de mínimo privilegio:* se refiera a la asignación de los privilegios mínimos necesarios a un objeto para garantizar el cumplimiento de sus tareas.

- *Principio de diseño abierto:* este es uno de los principios más importante y está dirigido a la no confidencialidad del diseño debido a que para garantizar la seguridad de algún mecanismo no es necesario que el diseño del sistema permanezca en secreto.

- *Principio de manejo y mitigación de riesgos:* una vez que se identifican los riesgos, definir mecanismos de mitigación es mucho más sencillo. Este es un tema que se estará tratando en un epígrafe más adelante.

- *Principio de prohibición por defecto:* está muy asociado al principio de *asignación de mínimo privilegio* y define que a menos que se le otorgue explícitamente permisos de acceso a un objeto, éste no debería tenerlo.

En este sentido, la implementación segura deberá partir de los siguientes principios que a continuación se proponen:

- *Correspondencia con el diseño:* se refiere a no implementar nada que no esté especificado en el diseño. Es decir, no crear nuevas estructuras, jerarquías de clases, objetos, etc., que no esté explicitado en el diseño.

- *Principio de no desbordamiento y chequeo de sintaxis:* estos son dos elementos fundamentales en la calidad del software. Por un lado la evaluación de los desbordamientos bien sea de memoria o de variables específicas dentro de un programa y por otro lado, la verificación de buen uso de los comandos o palabras reservadas en el lenguaje de programación.

- *Buen uso del control y manipulación de errores y excepciones:* no solo por el hecho de que el sistema debe desarrollar la "capacidad" de mantenerse estable ante una situación anómala o un error, sino que deberán implementarse los mecanismos que permitan interactuar con el usuario ante una de estas situaciones.

Estos son solo algunos elementos a tener en cuenta para garantizar la seguridad en un sistema desde el propio diseño. Avanzar en la búsqueda de elementos de la seguridad desde el diseño no es un aditivo del desarrollo de software sino una imperiosa necesidad.

2.1.5 INTERFAZ

Un momento importante luego que se definen las estructuras o elementos de software (clases, componentes, módulos, subsistemas, etc.) es la definición de mecanismos de comunicación entre éstos. Sin embargo, no de menos importancia, otros mecanismos como los que permiten al sistema integrarse con otros sistemas o viceversa, constituyen de igual manera uno de los temas a tener en cuenta para el desarrollo de un diseño lo suficientemente flexible, escalable, integrable y capaz de mantener su estructura ante los cambios.

Desde un punto de vista industrial, el competitivo mercado de software ha forzado a las compañías a buscar estrategias que les permita mantenerse dentro de la competencia. De esto se deriva la existencia cada vez mayor de sistemas que hacen lo mismo, por lo que son los elementos de calidad los que determinan la factibilidad y eficiencia de un determinado sistema.

Por otro lado está la cuestión de que mientras más se integre un sistema a otros ya existentes con el objetivo de compartir recursos y funcionalidad, muchas más probabilidades tiene éste de permanecer en el mercado.

De ahí la necesidad de lograr un alto grado de capacidad de integración del sistema incluso desde etapas tempranas como lo es el diseño.

2.2 PRINCIPIOS Y BUENAS PRÁCTICAS DEL DISEÑO ORIENTADO A OBJETOS DESDE UN ENFOQUE INDUSTRIAL

La industrialización del software data de hace unas décadas atrás. En Este sentido se podría mencionar la Compañía Hitachi, cuando en 1969 comenzó a adoptar el término fábrica de software por primera vez, al fundar Hitachi Software Works. Desde entonces han aparecido en el mercado mundial del software empresas con este modelo de producción como son las empresas europeas *Soluziona,* con el modelo de fábrica de software *Lleida; PSL Factory Europe* con *Murcia; Indra* con el modelo instalado en *Badajoz;* como otros casos recientemente incorporados a éste método de producción de software.

Algunas de las ventajas de industrializar la producción de software han sido anteriormente mencionadas. El abaratamiento de los costes de producción –de hasta un 20% de costes inferiores respecto al resto de instalaciones como la compañía *Cap Gemini*-; la facilidad de reutilización de elementos de software –como es el caso de la experiencia de la compañía *Indra*, la que dispone de un conjunto de herramientas y mecanismos para la reutilización del software-, son algunas de estas ventajas, sin embargo existen otras asociadas al fenómeno de la rotación de personal y al vertiginoso crecimiento del mercado de fábricas de software, estimado en un 10% -el doble que el resto de los modelos tradicionales del negocio del desarrollo, según el diario El País (Diario El País, 2008)-.

Dentro de todo este proceso de industrialización del software, el proceso de diseño ocupa un lugar sumamente importante. Los ingenieros dentro de éste, manejan sofisticados modelos de desarrollo, herramientas de programación, estándar y exhaustivos sistemas de control de proyectos que automatizan los procesos –especialmente de diseño, generación de código, gestión de proyecto, gestión de control de cambios, gestión de la calidad, entre otros-.

El diseño se ve desde una perspectiva funcional y es desarrollado desde la propia *casa*[9] del cliente, por lo que esa interacción entre equipo de desarrollo y cliente, es la base del éxito de un buen diseño.

Según un estudio publicado por el ya antes mencionado diario *El País,* en la compañía *Accenture* sólo se producen 15 errores por cada millón de líneas de código, y se reutiliza el 52% del software. Sus programadores hacen 2,3 líneas más de código por día que el desarrollo tradicional, y los plazos de entrega se respetan en el 99,02% de los casos. Este año 2008 espera lograr la mayor cota de calidad informática, la CMM 5.

Para desarrollar un proceso de diseño con altos niveles de calidad, desde un enfoque industrial, es preciso:

1- Contar con un equipo de diseñadores de software experimentados, bien organizado y conocedores del dominio del problema sobre el cual se estará haciendo la modelación del sistema.

2- Desarrollar un proceso de diseño jerárquico *hacia abajo*[10] en todo momento, manteniendo la trazabilidad para poder realizar *backtracking* cuando se ha generado algún problema en alguno de los niveles de detalles.

3- Análogamente, evitar la jerarquización del diseño de abajo *hacia arriba*[11] debido a que éste provoca en la mayoría de los casos, el rediseño del sistema.

4- El diseño deberá ser abierto en todo momento y bien documentado, haciendo posible la familiarización con éste de todo el equipo de trabajo.

---

[9] Desde su empresa o institución de trabajo.

[10] Es un enfoque de diseño, consistente en prestarle atención inicialmente a los aspectos globales e ir desarrollando el proceso hacia un nivel más detallado. Conforme el diseño progresa, el sistema se descompone en subsistema, poniéndosele mayor consideración a los detalles específicos.

[11] Lo contrario al enfoque *hacia abajo.*

5- Desarrollar y aplicar desde inicios algún mecanismo de pruebas conceptuales y estructurales al modelo de diseño, con el objetivo de garantizar la calidad del proceso y del producto.

6- Aplicar patrones de diseño y documentar el uso de éstos en el modelo de diseño.

7- El diseño deberá conducir a interfaces cada vez más sencillas y generales con el objetivo de que se reduzca la complejidad de conexión entre los módulos y el exterior.

8- Definir y aplicar mecanismos de detección temprana de errores para reducir los costes de corrección.

9- Implementar un mecanismo de revisiones sistemáticas del diseño por pares.

10- Centrar la atención en desarrollar un diseño en principios sencillo y claro, y abstenerse de garantizar cualquier tipo de *optimización* hasta que el modelo cumpla con determinada completitud y pueda ser analizada la posibilidad de mejorar algún aspecto, ya sea de rendimiento, la nitidez de las estructuras, entre otros elementos.

Estos son algunos elementos, técnicas y principios que permitirán desarrollar un proceso de diseño con calidad. En el *epígrafe 2.3* se estará proponiendo específicamente un procedimiento de buenas prácticas para el proceso de diseño, que expone de manera organizada las ideas anteriormente expuestas.

2.2.1 ELEMENTOS A TENER EN CUENTA PARA CONCEBIR LA MODULARIDAD DEL DISEÑO

La modularidad es la suma de dos aspectos prácticamente contrapuestos en el desarrollo de software: la cohesión y el acoplamiento; conocidos por algunos como *el yin y el yang* de la ingeniería del software. En realidad más que la suma de estos dos elementos, es la capacidad de combinarlos para lograr una mejor eficiencia.

Evidentemente, cuando se desea garantizar determinado nivel de cohesión, podría estarse sacrificando por otro lado determinado nivel de acoplamiento. Lo cierto es que dicho relativismo hace que la elección se base en las necesidades y/o intereses de clientes y desarrolladores.

Para lograr niveles aceptables de acoplamiento, la clave es realizar un diseño cada vez menos monolítico sobre la base de la descomposición del problema principal para reducir la complejidad y el esfuerzo de desarrollo e ir integrando paulatinamente cada *sub-diseño* –partes de diseño-. Sobre éste último elemento, la descomposición a la que se hacía referencia, nunca deberá ser ni muy reducida, para evitar que el diseño sea monolítico, ni muy amplia, para que el esfuerzo de integración no sea muy costoso.

2.2.2 LA REUTILIZACIÓN DEL DISEÑO DESDE UN ENFOQUE INDUSTRIAL

Como se había dicho en ocasiones anteriores, reutilizar –ya sean elementos de diseño o código – disminuye el tiempo de desarrollo y abarata los costos de producción, a lo que se le podría añadir que permite concentrarse en el dominio del problema y no en problemas que ya se encuentran evaluados, analizados e incluso solucionados por otras personas.

Un elemento importante para lograr efectividad y buen impacto organizativo en el diseño, consiste en institucionalizar la reutilización. Una idea que podría apoyarse es

la de incentivar la búsqueda de oportunidades para reutilizar desde una perspectiva coordinada.

La reutilización efectiva del diseño se alcanza mejor asignándoles ésta tarea a personas en específico, los que serían los responsables de garantizar la actividad concretamente además de extender el análisis hasta el impacto de dicha reutilización en la fase de implementación del sistema.

La reutilización debe de estar bien fundamentada, pues en un equipo de trabajo ésta puede impactar negativamente en el código de los otros integrantes, y por supuesto, para que esto no suceda, deben existir personas que tengan una perspectiva más general del proyecto que se realiza. Por ejemplo, un programador debe estar exonerado de esto, su función principal es codificar, claro que la comunicación entre los miembros que implementan y la de las personas encargadas del proceso de reutilización tiene que ser clara y estar definida de forma vertical y transversal, en ambos sentidos.

La reutilización sistemática requiere una organización apropiada y una cultura idónea. Se deben revisar las prácticas de administración, las estructuras de organización y las tecnologías utilizadas para explotar eficientemente los elementos reutilizables.

¿Reutilizar diseño implicará reutilizar código? Existe un nexo indiscutible entre los dos elementos. La diferencia recae en el nivel de abstracción en que se basan ambas situaciones. Primero, reutilizar diseño consiste básicamente en reutilizar modelos –como es el caso de los patrones de diseño para la solución de problemas recurrentes-, lo que está a un nivel de abstracción más elevado que reutilizar código, lo cual se refiere a emplear desde plantillas de desarrollo –como la experiencia de *deXma*[12]- hasta conjuntos de librerías de clases, *frameworks* –aunque en éste último caso se estaría hablando también de reutilización de diseño-, etc. La respuesta a esta pregunta es **no, reutilizar diseño no implica reutilizar código.** Sin embargo, en

[12] *deXma®* es un conjunto de bibliotecas, herramientas y servicios orientados al desarrollo de aplicaciones corporativas en .NET

muchos casos existirá la posibilidad dado a que cuando se reutiliza un diseño, o una sección de un diseño, si éste está previamente implementado, se debiera entrar en un análisis más específico relacionado con la similitud de escenarios para reutilizar el código de dicho diseño. Sencillamente la implicación es unidireccional. No tiene sentido reutilizar código si no se reutiliza el diseño, lo cual no se cumple necesariamente en sentido inverso.

Un segundo elemento importante para la reutilización y su correspondiente institucionalización, es la adopción de algún modelo que permita definir los procesos a seguir para alcanzar una mejor efectividad en esta actividad. La universidad de Valladolid ha desarrollado uno de estos modelos en el *Grupo de Investigación en Reutilización y Orientación al Objeto (GIRO)*, denominado *Modelo de Reutilización GIRO (MRG).* Este modelo es una alternativa para el desarrollo de software en el marco de la reutilización sistemática y con el soporte de estructuras complejas denominadas *mecanos* que están formadas por elementos de análisis, diseño e implementación (LÓPEZ, y otros, 2004).

Entre otros aspectos de marcada importancia en el modelo MRG, se encuentra la de comenzar a tener en cuenta la reutilización desde etapas tan temprana como son los requisitos funcionales del sistema a partir de la analogía de escenarios, debido a que éstos –los requisitos funcionales- dan la medida de si un elemento reutilizable se puede emplear en un contexto dado (GARCÍA PEÑALVO, y otros, 2004). Este último proceso consiste en recuperar un caso base con potencial de reutilización, derivar características aplicables al caso destino, aplicar características del caso base al caso destino y retroalimentar el repositorio de casos –o datos- (LÓPEZ, y otros, 2004).

Por último, para lograr un buen proceso de reutilización, es preciso crear una infraestructura tecnológica, basada en la definición e implantación de algún repositorio de elementos reutilizables, del que se puedan nutrir todos los miembros del equipo de desarrollo del diseño, para realizar los análisis correspondientes a esta actividad.

Por supuesto, el control del proceso de reutilización deberá constituir el eje central, para que el producto se obtenga con buena calidad. Este control estará explícito en el modelo seleccionado e implícito en el proceso de reutilización en cuestión.

Resumiendo, deberán tenerse en cuenta tres aspectos esenciales para lograr un adecuado proceso de reutilización en el diseño:

- Definir un equipo responsable de realizar esta actividad.
- Seleccionar un modelo de reutilización –el cual bien podría ser el propio modelo MRG-
- Crear un repositorio o biblioteca de elementos reutilizables para próximas situaciones.

Si bien no es el único elemento que debe tener un buen diseño, la reutilización y más aún, la reutilización institucionalizada, es uno de los más importantes atractivos del proceso de diseño a nivel industrial.

2.2.3 MECANISMO DE PERSISTENCIA

La persistencia es la capacidad que permite que la existencia de los datos trascienda en tiempo y espacio en un sistema informático (VÉLEZ SERRANO, y otros, 2005).

Este es un aspecto de suma importancia a la hora de diseñar cualquier software de cualquier índole. Evidentemente persistir información hace aparecer la necesidad de recuperarla en determinado momento. De esta manera sería conveniente orientar el esfuerzo a la definición de esquemas de persistencias debido a que éste deviene en un nicho de clases reutilizables cuya finalidad básica es la de brindar servicios a los objetos persistentes del sistema, bien sea en una base de datos, fichero u orígenes de datos –como Microsoft ODBC o cualquier otro-.

Pensar en definir un esquema de persistencia permitirá ganar en tiempo a la hora de construir un sistema además que en el contexto de la industrialización del proceso de

desarrollo de software, además de que explota una de las principales característica que debe tener todo proceso de desarrollo a escala industrial: la reutilización.

En este sentido el esquema de persistencia deberá definirse de tal manera que pueda ser extensible para diferentes medios de almacenamiento, para ganar en la posibilidad de que pueda ser empleado en la construcción de diversos sistemas.

Por otro lado, en vez de entrar en la definición del algún esquema de persistencia podría pensarse en el uso de algún motor de persistencia ya previamente definido capaz de *mapear* [13] las clases sobre algún soporte de almacenamiento de datos, tal es el caso del marco de trabajo *Enterprise JavaBeans (EJB)* para el lenguaje Java.

2.2.4 ROBUSTEZ Y FIABILIDAD DEL DISEÑO

Es evidente que un diseño robusto hará más fiable el sistema. Por un lado, la robustez del diseño radica en el grado de *capacidad* que presenta para *funcionar correctamente* frente a entradas de información erróneas. Es un término muy asociado a los niveles de tolerancia de errores y fallos y más usado en el contexto de la implementación. Sobre la fiabilidad del diseño se puede decir que es la *probabilidad* de que éste *"funcione"* tal cual se espera que lo haga bajo condiciones fijas y durante un tiempo determinado. En realidad, es preferible entender el término *fiabilidad* en el contexto del diseño, como la correspondencia y cumplimiento de éste con los requerimientos del sistema, dado a que la fiabilidad, como se mencionaba en el capítulo anterior, es un atributo que se mide en etapas posteriores y está relacionado más específicamente con la *fiabilidad del sistema*. Evidentemente desde etapas de diseño puede comenzarse a garantizar dicha fiabilidad, partiendo de mecanismos que a continuación se tratarán en más detalles.

---

[13] Relación entre una clase y su almacenamiento persistente (p.ej. una tabla de la BD), y entre los atributos del objeto y los campos (columnas) de un registro.

Temas muy relacionados con la robustez en el diseño son los aspectos a tener en cuenta a la hora de diseñar los mecanismo de recuperación y corrección de errores, fallas y defectos del sistema; cómo mantener la disponibilidad continua; cuáles estrategias deben seguirse para definir las políticas de tratamiento de excepciones y cómo estas políticas sean definidas acorde a los principios naturales del diseño; el uso de patrones y *frameworks* para automatizar los mismo; entre otros elementos.

Antes de continuar, es preciso delimitar las diferencias entre error, defecto y falla. En principios los tres términos corresponden a un mal funcionamiento del software aunque en sentido general, la palabra *error* suele ser empleada para tipificar todos y cada uno de los tipos de mal funcionamiento.

El *defecto* es cuando el sistema fue bien diseñado pero alguna parte de él –o su totalidad- fue mal implementada y por ende da al traste con un correcto funcionamiento.

La *falla* es una clasificación más asociada al mal funcionamiento del sistema en condiciones no previstas en el diseño, tal es el caso de la mala manipulación y tratamiento de excepciones. Por último, podría decirse que los *errores*, aunque siendo un concepto más general, se refieren al mal funcionamiento del sistema, de origen en principio desconocido.

En aras de lograr un mayor nivel de *fiabilidad* del diseño, pudiera aplicarse alguna técnica para el análisis de fiabilidad como por ejemplo la de construcción del Árbol de Fallos. El *Análisis del Árbol de Fallos (AAF)* es una técnica que consta de cuatro fases: *Definición del Sistema, Construcción, Análisis Cualitativo y Análisis Cuantitativo* (GALVÁN, y otros, 2005). Ésta es una labor muy compleja, por lo cual pudiera no ser conveniente o práctico utilizarla en el ámbito de la industria, sin embargo, es una opción que pudiese ser evaluada dado a que han existido esfuerzos por automatizar este proceso de análisis, haciéndolo cada vez más factible de aplicar.

Robustez y fiabilidad son dos características de un sistema con una alta calidad. En el contexto de la industria del software, son dos elementos que todo buen diseño deberá cumplir para lograr un elevado nivel de competitividad en el mercado internacional.

2.2.4.1 MECANISMOS DE RECUPERACIÓN DE ERRORES

Todo sistema debe implementar algún mecanismo de recuperación de errores, bien sea hacia adelanta o hacia atrás, que permita corregir el funcionamiento del mismo o llevarlo a un estado anterior de funcionalidad respectivamente.

La importancia de emplear sistemas que generan código se ve reflejada, entre otros aspectos, en la reducción de las probabilidades de que se produzcan errores de programación o implementaciones incorrectas del diseño. Con el surgimiento y desarrollo de sistemas generadores de códigos a partir de modelos de diseño, la probabilidad de errores por líneas de código ha sido reducida significativamente. Esto puede ser comparado con el análisis realizado por Hecth hace unas décadas, el que llegaba a la conclusión de que por cada un millón de líneas de código existen veinte mil errores de codificación.

En una entrevista realizada a la empresa venezolana desarrolladora de software E4GS, el 29 de abril de 2008 (*ver ANEXO 9*), su director, el Ing. José Yván Bohorquez, planteaba que el emplear ***software para generar software y automatizar procesos*** –como los procesos de diseño y codificación- es una de las prácticas que hace exitoso el proceso de industrialización del desarrollo de software, permitiendo reducir los costes de producción y el tiempo de desarrollo además de reducir considerablemente también los errores de codificación (BOHORQUEZ, 2008).

En el mundo existen diversos métodos para realizar la comprobación de software que posibilitan analizar las averías en términos de errores del sistema. Desde la definición de modelos en que se utilizan métodos bayesianos para hacer inferencias sobre los

fallos en el programa para dos versiones del modelo, dados los datos procedentes de la fase de comprobación (WIPER, y otros, 2003), hasta el uso de pruebas convencionales de caja blanca y caja negra.

Todo mecanismo de recuperación de errores debe tener incorporado algún mecanismo de depuración para que después de cada avería, se depure el programa y se ejecute de forma iterativa hasta obtener una fiabilidad aceptable.

Para aumentar la cobertura de detección de errores en un mecanismo de recuperación, podría definirse algún esquema de tolerancia de fallos.

Los fallos no pueden ser observados de forma directa, sino que deben ser deducidos a través de la presencia de errores. Dado a que un error es entendido como un estado del sistema, pueden realizarse comprobaciones para ver si existe o no el error. Idealmente, el mecanismo de detección de errores debe detectar –valga la redundancia- cualquier posible error causado por aquellos fallos que el esquema de tolerancia a fallos intenta tratar. Sin embargo esto no es totalmente posible. Para poder ganar en efectividad a la hora de realizar algún diseño de esquema de tolerancia a fallos, es preciso se tengan en cuenta dos aspectos importantes:

- que al esquema o sistema de comprobación de tolerancia a fallos no debe afectarle los fallos del sistema que se pretende comprobar,

- y en segundo lugar, el esquema debe ser completo y preciso, de manera que pueda tener en cuenta todos o la mayor cantidad posible de errores y que nunca informe sobre un error inexistente, respectivamente.

Existen dos maneras de recuperación de un sistema dada la ocurrencia y detección de algún error. Por un lado está la técnica de recuperación hacia adelante (*fordward recovery*), la cual se encarga de realizar un análisis tanto del alcance de los daños como de las razones de los mismos para poder eliminar el error del sistema,

llevándolo a un estado de normalidad. Esta técnica realiza un importante empleo del mecanismo de tratamiento de excepciones como vía para realizar la recuperación del sistema hacia adelante, haciendo correcciones desde un estado erróneo.

Por otra parte está la técnica de recuperación hacia atrás (*backward recovery*), la cual se encarga de llevar el sistema a un estado anterior, el que se supone sea correcto. Esta técnica conlleva a un considerable consumo de recursos debido a que debe almacenarse periódicamente el estado del sistema, lo que la hace en ocasiones ser inapropiada para sistemas done el rendimiento sea un requisito indispensable para el correcto funcionamiento. Sin embargo, su dependencia a la naturaleza de la falla la hace apropiada para fallos transitorios.

Se ha estado hablando de dos conceptos fundamentales: la prevención y la tolerancia a fallos. La primera se realiza en cuatro fases fundamentales:

1- Evitación.
2- Comprobación.
3- Detección.
4- Eliminación.

La ***evitación de fallos*** trata de impedir que se introduzcan fallos durante la construcción del sistema para que no se produzcan errores posteriores. Este proceso se realiza desde dos perspectivas diferentes pero estrechamente relacionadas. Desde una perspectiva de hardware, mediante la utilización de componentes fiables, técnicas rigurosas de montajes de sub-sistemas, apantallamiento de hardware, entre otros aspectos. Y desde un enfoque de software, en la especificación rigurosa de requisitos, métodos de diseño comprobados, utilización de lenguajes de abstracción y modularidad.

Evidentemente tratar de evitar los fallos no garantiza que éstos no sucedan; por este motivo, la etapa de ***comprobación de fallos*** es crucial para lograr una correcta efectividad en el proceso. Las revisiones de diseño –o revisiones técnicas

(PRESSMAN, 2001)- es una de las técnicas de comprobación más usadas y efectivas, además de la verificación de programas y la inspección de código. Otra técnica es la de realizar pruebas al sistema. Dichas pruebas son necesarias pero no suficientes ya que presentan algunas limitantes, como la de no ser exhaustivas; muestran que hay errores pero no que no los hay, lo que las hace un mecanismo inapropiado para afirmar la correctitud del sistema además de que en ocasiones es muy difícil reproducir las condiciones reales en las que operará el sistema.

Podría decirse que la ***detección de fallas*** forma parte de la comprobación. En realidad estas etapas se complementan unas con otras. El objetivo de separar parcialmente la detección, consiste en que esta actividad lleva asociado un análisis y generación de documentación de las fallas encontradas, lo que permite realizar una correcta toma de decisiones para pasar a la última etapa.

La ***eliminación de fallos*** es la última etapa y una de las más importantes. La efectividad de ésta depende de la eficiencia con que se hayan desarrollados las otras etapas anteriores.

Para el caso de la tolerancia a fallos, ésta trata de que el sistema continúe funcionando aunque se produzca algún fallo. Existen diferentes grados de tolerancia a fallos en dependencia del tipo y objetivos de la aplicación.

Buscar que el sistema continúe funcionando, al menos durante un tiempo, sin perder funcionalidad ni prestaciones, hace que el grado de tolerancia sea ***completo***. En cambio, si el sistema sigue funcionando con una pérdida parcial de funcionalidad o prestaciones, hasta la reparación de la falla, se estaría en presencia de un grado de tolerancia de ***degradación aceptable*** o de ***tolerancia parcial***. Para el caso en el que la respuesta del sistema ante una falla sea parar sus prestaciones o funcionalidades garantizando una integridad del entorno y la información, se podría decir que el grado de tolerancia sería ***parada segura*** (PUIG, y otros, 2004).

Para aumentar la fiabilidad del sistema pueden usarse los dos mecanismos para crear uno solo, un mecanismo de prevención y tolerancia a fallos.

2.2.4.2 POLÍTICAS DE TRATAMIENTOS DE EXCEPCIONES

Desde los inicios de los lenguajes de programación, incluso, desde antes de paradigmas como el orientado a objetos, la gestión de errores ha sido uno de los asuntos más difíciles.

Resulta tan complejo diseñar un buen esquema de gestión de errores, que muchos lenguajes simplemente lo ignoran, delegando el problema en los diseñadores de la librería, que lo resuelven a medias, de forma que puede funcionar en muchas situaciones, pero se pueden eludir normalmente ignorándolos.

El problema más importante de la mayoría de los esquemas de gestión de errores es que dependen de que el programador se preocupe en seguir un convenio que no está forzado por el lenguaje. Si los programadores no se preocupan, cosa que ocurre cuando se tiene prisa, esos esquemas se olvidan fácilmente.

La gestión de excepciones forma parte del mecanismo de gestión de errores de un sistema. La diferencia radica básicamente en que la primera "*conecta*" la gestión de errores directamente con el lenguaje de programación y a veces hasta con el sistema operativo. Una excepción es un objeto[14] que se "*lanza*" desde el lugar del error y puede ser "*capturado*" por un *manejador de excepción* apropiadamente diseñado para manipular este tipo en particular de error.

Podrían citarse innumerables normas para el tratamiento de excepciones, sin embargo ya hay una cantidad considerable de trabajos que brindan un enfoque más amplio y detallado sobre este tema, como es el caso de los artículos del EHOOS-

[14] Entiéndase objeto desde la perspectiva de paradigma orientado a objeto. Refiérase a la instanciación de alguna clase en particular dentro de algún esquema de clases que defina el mecanismo de tratamiento de excepciones del sistema.

Workshop (Exception Handling in Object Oriented Systems) desarrollado con motivo del evento ECOOP´2003, efectuado por la Universidad Técnica de Darmstadt en Alemania, donde especialistas de la IBM, la SD&M Research, entre otras corporaciones, expusieron sus experiencias, teorías y enfoques sobre lo relacionado con el manejo de excepciones en sistemas orientados a objetos (SIEDERDLENBEN, 2003), las aserciones en Java (*assert*) (REIMER, y otros, 2003), mecanismos de manejo de excepciones (MILLER, y otros, 2003) y demás.

La esencia de una correcta política de manejo de excepciones en la construcción de un sistema es ***desarrollar diseños simples, modulares y extensibles***. El mayor enemigo de la fiabilidad de un sistema es la complejidad. Una codificación elegante y legible, así como la claridad y simplicidad de las construcciones, ayuda a que se eleve la fiabilidad del software. Tanto el diseño como el código deben estar "escritos" para ser "leídos".

Sobre este último aspecto, una importante observación: ***es preciso mantener un respeto a los principios, reglas y criterios de la modularidad***. Tanto el lenguaje como el método de diseño deben responder a estos principios. Es una necesidad que las políticas de tratamiento de excepciones sean definidas acorde con los principios naturales del diseño.

Para enriquecer el mecanismo de tratamiento de excepciones de un sistema, pudiera modelarse el diseño del mismo desde una perspectiva de teoría de aserciones que incluya todo lo relacionado con la declaración formal de lenguajes y diseño desde el punto de vista de las precondiciones de cada contrato de abstracciones, post-condiciones e invariantes, que viene siendo los posibles valores a asumir por parte del dominio de datos que se modela.

Un *aserto* es una sentencia que permite probar ciertas suposiciones sobre el programa y que deviene en unidad básica de dicho mecanismo de aserciones. Cada *aserto* contiene una expresión *booleana* que se supone que será cierta cuando el aserto se ejecute, de lo contrario el sistema lanzará una excepción. Una vez que se

verifique que la expresión booleana es realmente cierta, el aserto confirma las suposiciones sobre el comportamiento del programa aumentando a su vez la confianza de que esté libre de errores.

Utilizar asertos como invariantes de clases es otro modo de hacer más fiable el mecanismo de tratamiento de excepciones. Una *invariante* de clase es un tipo de aserto que debe verificar cualquier instancia de una clase en cualquier momento, excepto cuando se está realizando una transición de un estado consistente a otro.

Sea cual sea el mecanismo que se vaya a emplear, debiera partirse del principio de que ***el tratamiento de excepciones no forma parte de la lógica de funcionamiento del sistema, sino de la vía para hacer que éste funcione correctamente y de manera fiable y robusta***. Es decir, cuando se comienza a diseñar un sistema, evidentemente se parte de los requisitos funcionales y no funcionales que deberá cumplir y tener -respectivamente- éste, por lo que el manejo de excepciones, en conjunto con otros elementos, como lo son la manipulación de la memoria, rendimiento, validaciones para acceso a bases de datos, entre otros, forma parte de *aspectos* que *entrecruzan* la lógica de negocio del sistema y que de alguna manera u otra, hacen más engorrosa la revisión, entendimiento, mantenimiento y reutilización del diseño y el código.

Partiendo de este último problema, se plantea como solución más factible la de aplicar una metodología de diseño soportada por el *paradigma de orientación a aspectos*. Es decir, introducir la Programación Orientada a Aspectos (*AOP*, del Inglés *Aspect Oriented Programming*) en el desarrollo del sistema en cuestión. Esto trae consigo varias ventajas, una de ellas es que permite la asignación de tareas específicas a grupos de diseños, en el sentido de que por ejemplo, una parte se encuentre diseñando la lógica que deberá cumplir el sistema y la otra, de manera "aislada", diseñando mecanismos que no tienen directamente nada que ver con dicha lógica, como es el caso del tratamiento de excepciones. Evidentemente no es la única ventaja que traería consigo la aplicación de este tipo de paradigma. Su positiva

repercusión iría incluso hasta la facilidad de planificación y control, legibilidad del diseño y el código, minimización en los tiempos de desarrollo y los costes de integración gracias al empleo de *frameworks* y lenguajes especializados en el manejo de *aspectos,* así como otras ventajas asociadas al rendimiento, mantenibilidad, seguridad, escalabilidad, extensibilidad, reutilización y fiabilidad del diseño.

En resumen, los mecanismos de tratamiento de excepciones son, más que necesarios, imprescindibles en todo sistema, pero embeberlos en el diseño y el código de la lógica de funcionamiento básica del sistema, que debe responder al cumplimiento de requisitos funcionales del mismo, sencillamente es una mala práctica que trae consigo consecuencias en el mantenimiento del modelo de diseño y el código producto a la poca legibilidad; además del comprometimiento de la fiabilidad de los mecanismos de detección de errores debido a que al producirse una falla sería muy difícil focalizar la región en dónde se produjo y hacer más difícil conocer la causa.

2.2.4.3 ESTRATEGIAS DE MANTENIMIENTO

Existe una incorrecta tendencia de denominar *mantenimiento* a lo que se debiera llamar *evolución.* Esté previsto o no, los diseños de sistemas, bien sean sistemas simples o complejos, tienden a evolucionar con el tiempo, situación que con frecuencia se etiqueta de manera errónea como *darle mantenimiento al diseño.* Evidentemente este análisis es análogo para con los sistemas en sentido general.

Según Grady Booch (BOOCH, 1998) "*es mantenimiento cuando se corrigen errores y evolución cuando se responde a requerimientos que cambian".* Para ser más preciso habría que añadirle a esta definición, cuyo enfoque correctivo limita hasta cierto punto la actividad de dar mantenimiento al sistema, que aspectos como el rendimiento, la adaptabilidad al entorno en que opera el sistema, además de los ya mencionados errores, son otros de los aspectos que hacen del mantenimiento una actividad más

abarcadora. Incluso, algunos autores le dan a éste una visión evolutiva, haciendo posible que la brecha entre un concepto y otro –mantenimiento y evolución- sea solapada.

Si se fuera a comprar la actividad de darle mantenimiento al software y darle mantenimiento al hardware habría que entrar en detalles que marcan la diferencia considerablemente y más aún para el caso de la complejidad de dicho mantenimiento. Por una parte el hardware comienza a presentar fallos a inicios, quizás producto a defectos de diseño de fabricación, hasta que se logra corregir estos defectos y la tasa de fallos cae, idealmente, a un nivel estacionario. Conforme pasa el tiempo dichos fallos comienzan a reaparecer pero para este entonces producto a que éste se estropea. Si embargo el software no se estropea, sino se deteriora. Aunque parezca contradictorio, la diferencia consiste en que durante su vida, el mantenimiento que se realiza al software va introduciendo nuevos defectos haciendo que la curva de fallos tenga picos aún cuando ésta pueda mantenerse en un estado aparentemente estable. El incremento del nivel mínimo de fallos va haciendo que el software se vaya deteriorando. Par el caso del hardware, cuando se estropea, se coloca otro de repuesto y el problema se soluciona parcialmente; sin embargo para el software no hay "piezas de repuestos", lo que hace considerablemente más compleja la actividad de dar mantenimiento al software.

La facilidad de mantenimiento del diseño es dependiente de la calidad de éste. Esta dependencia es mucho más acentuada para el caso de los atributos de modularidad del diseño, el acoplamiento, la flexibilidad y la cohesión. Además de esto, el mantenimiento tributa a la gestión de costes del sistema. Debe ser definida una política de mantenimiento adecuada y orientada a optimizar costes. En este caso, dichos costes de mantenimiento requerido para alcanzar un cierto nivel de confiabilidad (y por lo tanto seguridad y producción a largo plazo) están balanceados con los costes de los fallos (AMENDOLA, 2003). Gran parte del coste asociado al mantenimiento del software puede ser optimizado a través de un diseño flexible y estable frente a futuras modificaciones (CABRERO, y otros, 2008), por esta razón es

una buena práctica comenzar a pensar en el mantenimiento desde los mismos inicios del diseño, en el sentido de minimizar estos costes.

La principal estrategia a seguir deberá ser la de realizar ***mantenimiento preventivo sistemático,*** con el objetivo de dar respuestas tempranas a señales de deterioro. El mantenimiento es una actividad muy ligada a otras acciones del diseño, como lo son la gestión de riesgos, mecanismos de pruebas, mecanismos de recuperación de errores y manejo de excepciones, revisiones técnicas del diseño, gestión de cambios, entre otros aspectos.

Es sumamente importante que el personal de mantenimiento del diseño esté a tono con los estándares usados, las políticas de manejo de excepciones y la aplicación de los conceptos de fiabilidad durante la fase de diseño. Dicho personal bien puede ser parte del equipo que realizó la sección del diseño objeto de mantenimiento u otras personas designadas a efectuar mantenimiento pero que hayan estado en contacto directo con el proceso de diseño del sistema. No es conveniente reasignar personal nuevo para que ejecute esta actividad, dado que aumentarían las probabilidades de introducir nuevos errores y aumentarían el tiempo de respuesta a cambios; dos factores que evidentemente afectarían la productividad de una empresa de software.

Otra buena práctica es la de la ***cuantificación de las decisiones de diseño en términos de costes de mantenimiento***. Esto quiere decir que dado a que detrás de cada decisión de diseño, se ponen en juego una gran cantidad de dinero y oportunidades de mercado, es necesario que se optimice desde el punto de vista del *valor*, el mantenimiento implicado en cada decisión de diseño realizada. Existen innumerables esfuerzos por lograr este objetivo, sin embargo ningunas de las heurísticas, patrones o principios hacen posible este hecho (CABRERO, y otros, 2008).

Mary Shaw en su presentación inaugural de la conferencia internacional IEEE EQUITY 2007 expresaba: "*Necesitamos mejores maneras de analizar un diseño*

software y predecir el valor que su implementación ofrecerá a los clientes y desarrolladores" (SHAW, y otros, 2007).

Según Daniel Cabrero (CABRERO, y otros, 2008), muy a pesar de que se han realizado algunos esfuerzos, no existen propuestas que estén encaminadas a cuantificar el mantenimiento del diseño de un sistema, lo cual es muy cierto y lamentablemente sensible para este proceso dado a que es la fase más costosa del ciclo de vida del software (PRESSMAN, 2001).

Para hacer más óptimo la mantenibilidad del diseño, debe partirse básicamente de un ***diseño simple***. Eh aquí de nuevo la importancia de la simplicidad del diseño. Comenzando por modelar la solución de manera simple e incrementando la complejidad solo en aquellas partes del diseño donde el sistema lo requiera, el diseñador deberá ser capaz de evaluar la factibilidad de cada decisión de diseño desde un enfoque orientado a la mantenibilidad.

Finalmente se sugiere partir de algún modelo de reglas para la mantenibilidad del diseño, capaz de brindar un esquema que le permita al diseñador guiarse por él para realizar el análisis de factibilidad de algún cambio en el diseño y conservar a la vez los costes de mantenibilidad bajo control. De acuerdo al estudio realizado, se propone tomar como modelo el catálogo de *reglas de diseño* desarrollados por los miembros de la IEEE, J. Garzás y M. Piattini, publicado en el volumen 22 de 2005 de la IEEE con título "*An ontology for micro-architectural design knowledge*". Básicamente, el objetivo de este catálogo de reglas de diseño es presentar un conjunto de reglas que toda aplicación deberá cumplir para minimizar los costes de mantenimiento del diseño. La idea es que allí donde una regla sea violada, hay una posibilidad de mejora de diseño para hacer el sistema más estable y flexible frente a posibles cambios y ampliaciones.

Como es lógico, cada una de las decisiones de diseño estará respaldada por una justificación basada bien sea en la importancia de realizar el cambio o en el impacto –negativo o positivo- que éste pueda o no tener para el sistema en sentido general.

Partiendo de la generalidad de que el cliente no conoce el diseño, habría que pensar en alguna manera de poder identificar el *peso de importancia* de éste y no necesariamente desde la perspectiva del diseñador. Una buena práctica podría ser a partir de los requerimientos del software. Empleando criterios valorativos del propio cliente, podría identificarse de manera estimativa, el nivel de importancia que tiene para él determinado requerimiento, y empleando algún mecanismo e trazabilidad, llegar a trasladar dicho criterio a las partes del diseño que se encuentran en el abanico de análisis para la factibilidad de aplicar o no algún cambio.

De manera conclusiva, el mantenimiento no es algo trivial dentro del proceso de diseño, sino todo lo contrario. Constituye una labor cuya atención desde inicios del proceso y la aplicación de buenas prácticas, reglas, principios, un trabajo bien organizado, buena gestión y estimación de riesgos, entre otros aspectos relacionados con la gestión de proyectos, harán que su aplicación sea efectiva y el impacto en el sistema positivo.

2.2.5 EQUIPO Y ROLES

Este es un epígrafe de novedosa trascendencia debido a que nunca se ve al proceso de diseño desde una perspectiva hombre-equipo. Temáticas relacionadas con la gestión organizacional; la manera de crear un equipo de diseño; el cómo realizar las discusiones dentro de dicho equipo y al calor del desarrollo del diseño y cómo establecer y controlar estándares, generalmente son absorbidas por la definición de alguna metodología de desarrollo de software en sentido general o en otros casos sencillamente obviadas o mal empleadas.

Hay que partir de que el diseño es un proceso intelectual por lo que es producido por el talento de su creador o de sus creadores (PALACIO BAÑARES, 2008). Esta razón hace que los modelos que existen permitan que el diseño sea documentado de

determinada manera, comunicado de determinada manera y actualizado de determinada manera, pero no garantizan que éste sea bueno.

2.2.5.1 TRANSICIÓN DE GRUPO A EQUIPO DE DISEÑO

La ***experticia y conocimiento del dominio del problema de los miembros de un equipo de diseño*** influye, casi de manera determinante, en la obtención de un *buen* diseño. Por una parte la *experticia* de los diseñadores deberá estar basada en la experiencia que tengan éstos sobre cómo realizar el proceso, el empleo de patrones, métricas, lenguajes de modelado, paradigmas, estándares, tecnología y mecanismos. A esto deberá sumársele el conocimiento que tengan sobre el dominio del problema a modelar, los requerimientos del sistema, los estándares que se deberán emplear para este caso, el modo de trabajo del equipo, entre otros aspectos relacionados con la organización.

Lo anterior impone que se defina algún ***método de diagnóstico y formación del equipo de diseñadores***. El equipo de diseñadores funciona como una unidad organizacional auto-gestionada de manera parcial, debido a que no es una estructura organizativa ajena al proyecto. En este sentido, antes que el equipo inicie un programa de formación, debe hacerse un estudio minucioso con el fin de diagnosticar la naturaleza de los problemas del mismo. Esto ayudará a determinar cuáles son los cambios necesarios y si la formación será una actividad apropiada. En caso que así sea, se deberán centrar los temas de preparación en los problemas detectados en el diagnóstico, el cuál a su vez deberá estar orientado a las necesidades de la organización.

Por otro lado está la ***estimulación y no la competitividad*** dado a que esta última tiende a disminuir la confianza interna del equipo. En el contexto económico o del comercio, las empresas necesitan permanecer delante de sus competidores para poder sobrevivir en el mercado, sin embargo, esto no es un concepto que deba

aplicarse a los miembros del equipo puesto a que es una cuestión de ***sinergia y colaboración*** y no de "¿quién es mejor que quién?". En cambio, la incentivación o estímulo a la auto-preparación individual orientada a las necesidades de la organización, traería resultados positivos reflejados en la capacidad de respuesta de los miembros, la efectividad en la toma de decisiones y la correcta aplicación de los mecanismos, modelos, estándares o lenguajes.

Lo anterior da pie a otro importante análisis: ***la asignación de roles no debe contemplarse como un estímulo al desempeño sino dirigido desde una perspectiva de gestión y administración del conocimiento.*** Como antes se mencionaba, la competitividad, de manera declarada entre miembros del equipo, hace que cuestiones tan elementales como la cohesión, la sinergia, la cooperación y otros aspectos que tributan a la ***unidad*** de los miembros, se vean afectadas. Esto conlleva a la polémica de ¿cómo hacer que la auto-preparación no se convierta en competencia? Y como si fuera poco ¿cómo hacer que los miembros del equipo cuenten con una preparación competitiva? Para el caso de asignar algún rol a un miembro, es importante que sea sobre la base de dos aspectos importantes, los subjetivos (al amiguismo, compadrazgo, etc.) y objetivos (relacionados con su conocimiento, su disposición, responsabilidad, capacidad de comunicación).

La ***responsabilidad se orienta desde una perspectiva estrictamente individual a un enfoque individual-colectivo***. De este modo, muy a pesar que cada cual en el equipo de diseño tendrá su respectiva responsabilidad, todos en su conjunto deberán entender la necesidad de que el cumplimiento colectivo depende del cumplimiento individual. Esto podría llamarse sentido de pertenencia de las responsabilidades del equipo.

La ***comunicación*** es otro elemento a tener en cuenta a la hora de definir el equipo de diseño. Ésta debe realizarse de manera correcta, sistemáticamente y de fácil rastreo. La correctitud de dicha comunicación se debe basar en la colectividad y empleando algún medio efectivo, que sea de uso colectivo y preferentemente

personado –no impersonal-, lo cual no quiere decir que para lograra operatividad se usen mecanismos electrónicos como e-Mail, *chats*, mensajes instantáneas, etc., pero para lograr una mejor efectividad, personalizar este mecanismo, es decir, comunicar mediante pequeñas reuniones operativas o mítines, es mucho más efectivo. A esto se le puede sumar, que para lograr una mejor representación de la información, se pueden emplear mecanismos visuales –en cualquier soporte- que hagan más entendible, además de perdurable, la idea de lo que se comunica. Esto es más aplicable para los planes de trabajo, asignación de tareas, notas sobre cambios inmediatos, etc.

Las ***discusiones entre miembros del equipo sobre temas de trabajo deberán generar acuerdos*** y siempre deberán estar dirigidas por alguien. Los acuerdos que generen cambios deberán tener un tratamiento especial dado a que podría impactar –positiva o negativamente- en los costes de desarrollo, tiempos de entrega, entre otros elementos que determinan la calidad de todo el proceso y el producto.

Hasta aquí, algunos de los aspectos que deberán tenerse en cuanta a la hora de crear un equipo de diseño dentro de un proyecto de desarrollo de software.

2.2.5.2 ESTRUCTURA DE ORGANIZACIÓN

Generalmente la metodología de desarrollo es quien define la estructura organizativa de un proyecto, sin embargo las especificidades sobre esta estructura quedan a la merced de la adaptación que se haga a dicha metodología en el entorno del proyecto donde se empleará.

Para el caso del proceso de diseño existen incluso metodologías que no definen estructura organizativa concreta, es el ejemplo de *Extreme Programming (XP)*, cuyo proceso de diseño es llevado a cabo por equipos de programadores, embebiendo la responsabilidad de realizar el diseño en la responsabilidad de implementar el sistema, quizás por la misma *"agilidad"* con que se realiza el desarrollo del software.

El objetivo central de este epígrafe es definir una estructura organizativa general de lo que podría ser un equipo de diseño, incluso muy independientemente de la metodología de desarrollo que se pretenda usar.

Para ganar en flexibilidad sin perder en especificidad sobre esta definición de estructura de organización, es preciso se entienda que la misma puede ser adaptada de acuerdo a las condiciones reales del entorno y las necesidades concretas del proyecto.

A continuación se proponen los que podrían llamarse *departamentos de sub-procesos de diseño,* cuya definición se soporta de una determinada analogía con la estructuración general del proceso de desarrollo:

- **Gerencia general del proceso de diseño (GGPD):**

  La gerencia general del proceso de diseño es la ***que se encarga de ir guiando el proceso en sentido general***, análogo a la dirección general de la organización. Esta correspondencia en principios facilita la comunicación entre el equipo de diseño y el entorno de desarrollo en sentido general.

  La asignación de las tareas del equipo, organizar y dirigir las reuniones de control, definir estándares de diseño y realizar el diseño arquitectónico del sistema, son sólo algunas de las actividades que deberán realizarse por dicho departamento.

- **Análisis y Factibilidad Tecnológica (AFT):**

  El análisis de la correspondencia entre los artefactos de diseño y los que tributan a éste etapas tempranas del desarrollo de software, constituye el elemento de transición más importante al proceso de diseño. Entiéndase esto como la interpretación de diagramas de clases del análisis, diagramas de

colaboración entre objetos y clases del análisis, entre otros aspectos, que hacen posible la realización "física" del diseño.

Por otra parte está el ***análisis de la tecnología a emplear para el desarrollo del diseño***. Esto se refiere a las herramientas que se encargarán de automatizar el proceso, así como también aquellas para la modelación gráfica y la generación de código.

Finalmente, un elemento aún más importante, es el ***análisis sobre el compendio de patrones de diseño que podrían emplearse en la solución***, dependiendo de la tecnología, paradigmas y los lenguajes que se pretendan usar en la organización –o proyecto- para desarrollar el sistema. Será objetivo de esta actividad obtener una caracterización de los patrones que puedan ser empleados en la construcción del modelo de diseño.

Una vez definida la lista del conjunto de patrones que puedan ser compatibles con la tecnología y los paradigmas que rigen la dinámica del desarrollo, deberá pasarse a análisis posteriores que permitan acotar a solo aquellos que sirvan para dar solución específica a una problemática determinada y respondiendo en todo momento a los atributos de calidad del diseño.

- **Reutilización y Refactorización (RR):**

La reutilización es uno de los aspectos del diseño que deberán institucionalizarse para lograr efectividad en su aplicación. En este *departamento* se realizará un ***análisis sobre la posibilidad de reutilizar diseño dentro del propio proceso o de otros modelos externos, además de la capacidad que tenga el actual para ser posteriormente reutilizado***.

Por otro lado, ***se realizarán refactorizaciones continuas del diseño*** con el objetivo de mantener el mismo lo más correcto posible. La refactorización del diseño desde etapas tempranas, hace que el diseño evolucione, disminuyan

los fallos, aumente la calidad del producto y haga de la refactorización de código una actividad mucho más sencilla.

Las refactorizaciones deberán ser en todo momento lo más sencillas posibles, evitando que se modifique la lógica del diseño y que sean menores los riesgos de que el sistema deje de funcionar correctamente.

Refactorizar no debe convertirse en una actividad excesiva. Ir al detalle del diseño no debe ser un obstáculo para el desarrollo del proceso. No puede ser que se dedique más tiempo en refactorizar que en construir el diseño. La refactorización no debe comprometerse con el excesivo criterio personal de quien la realiza puesto a que la calidad del diseño a la larga termina comprometida, por esta razón es necesariamente una actividad controlada. La misma tiene una muy estrecha relación con el control de la calidad del diseño, el proceso de mantenimiento y el empleo de patrones de diseño.

- **Diseño de la Lógica Funcional del Sistema (DLFS):**

Este es uno de los principales departamentos del modelo propuesto debido a que es en donde se desarrolla la modelación que responde a toda la funcionalidad que deberá tener el sistema. Evidentemente existe una trazabilidad desde los requerimientos funcionales especificados por el usuario hasta este punto en cuestión, de modo que podrá realizarse *backtracking* en cualquier momento deseado.

Otra actividad importante de este departamento es la ***definición del prototipo de interfaz de usuario del sistema***, al que como es lógico, servirá para la comprobación del cumplimiento de los requerimientos del usuario.

- **Mantenimiento de Diseño:**

  El mantenimiento del diseño es una de las labores más costosas del proceso. Ésta se estará realizando desde principios y partiendo de un enfoque preventivo y sistemático.

  Como se estará abordando más adelante, la interrelación de este departamento con el resto deberá ser sumamente marcada puesto a que partiendo de los análisis que aquí se hagan podrán tomarse importantes decisiones en el proceso relacionadas con los costes y la planificación.

- **Planificación y Gestión de Calidad (PGC):**

  Tal y como el nombre lo indica, aquí se realizará y analizará la planificación y el cumplimiento de las actividades del diseño respectivamente.

  El objetivo no es realizar una planificación independiente a la que a nivel de organización o proyecto se haga, sino ajustar a las fechas de ésta las nuevas actividades que se vayan a desarrollar durante el proceso de diseño, tales como reuniones de control, las revisiones técnicas al diseño, planes de mantenimientos, etc.

  Por otro lado está la gestión y evaluación de la calidad del diseño como proceso y como producto. Desde este enfoque se deberán aplicar métricas de calidad relacionadas con el diseño y orientadas a la medición de los atributos de calidad del mismo.

  Aunque no se especifique en el nombre del departamento, la documentación de todo el proceso será otra de las tareas a realizar. Partiendo de esto, el *Expediente del Diseño*, como artefacto general del proceso, será archivado y gestionado por el personal miembro de esta sub-estructura organizativa

aunque su creación correrá a cargo de la gerencia general del proceso de diseño.

Hasta este entonces solo se han podido definir los departamentos pero no la manera en que éstos se relacionan y cooperan entre sí, así como tampoco los roles de trabajo asociados a cada uno de éstos, ni cómo fluye la información entre ellos. Antes de pasar a ese paso, es preciso que se realice una **importante observación:** *esta definición estructural queda al análisis de la organización para su factibilidad o no de aplicación. Es una estructura abierta a adaptaciones, ya sean las orientadas a la fusión entre departamentos, el empleo o no de departamentos o la alteración o no de la manera en que ellos interactúan.*

La *figura 2.2.5.2.1* muestra la propuesta de modelo de proceso de diseño basado en departamentos.

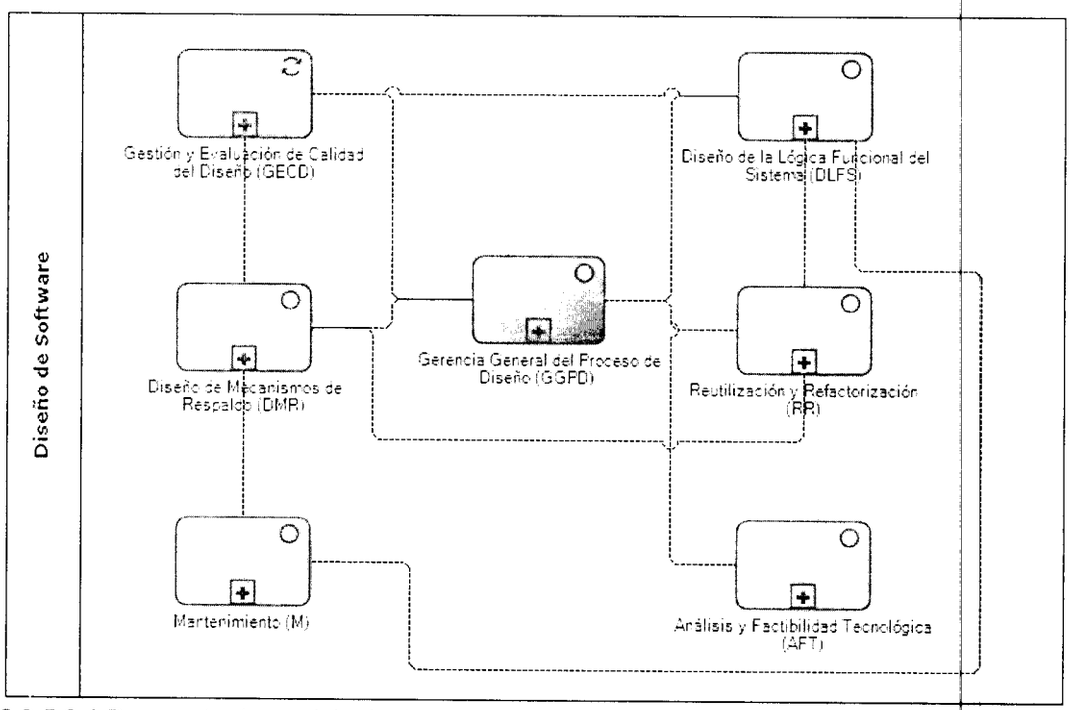

Fig. 2.2.5.2.1 Propuesta de modelo de de proceso de diseño basado en departamentos (desarrollado empleando *BizAgi Process Modeler*)

2.2.5.3 ROLES DE TRABAJO

La asignación de roles en un equipo de desarrollo naturalmente responde a la definición de éstos en la metodología que se esté empleando. Para el caso del diseño de software, son solo algunas las metodologías las que especifican roles encargados de realizar el proceso exclusivamente, tal es el caso de RUP. Sin embargo, metodologías como XP no definen ningún rol para el desarrollo del proceso de diseño, sino que esta tarea se encuentra implícita en las responsabilidades de los programadores.

El presente epígrafe tiene como principal objetivo definir un conjunto de propuestas de roles para el desarrollo del proceso de diseño, partiendo de la definición anterior sobre la estructura organizacional basada en departamentos. De la misma manera que dicha estructura, la asignación o no de estos roles estará dependiendo de los objetivos e intereses de la organización o proyecto.

La definición y distribución de roles correspondería de la siguiente manera:

- **Gerencia General del Proceso de Diseño (GGPD):**
  - Diseñador Principal.

- **Diseño de la Lógica Funcional del Sistema (DLFS):**
  - Diseñador de la Lógica Funcional.
  - Diseñador del Modelo de Datos.

- **Diseño de Mecanismos de Respaldo (DMR):**
  - Diseñador de Mecanismos de Respaldo.
  - Responsable de Integración.

- **Mantenimiento de Diseño (MD):**
  - Responsable de Mantenimiento.
  - Revisor Técnico.

- **Reutilización y Refactorización (RR):**
    - Analista de reutilización.
    - Responsable de Repositorio de Reutilización.
    - Responsable de Refactorización.

- **Gestión y Evaluación de Calidad del Diseño (GECD):**
    - Responsable de calidad del diseño.

- **Análisis y Factibilidad Tecnológica (AFT):**
    - Analista de Factibilidad Tecnológica.

Si bien es cierto que la definición de todos estos roles para el desarrollo del proceso de diseño constituye un elemento de soporte para éste, no puede ser que se convierta en un obstáculo que frene o haga más engorroso la realización del diseño. Por esta razón, es conveniente, como se ha venido haciendo énfasis anteriormente, que sea la organización la que decida qué roles son más convenientes, sobre la base de lo que aporte al propio desarrollo y si lo considera preciso, un miembro podrá desempeñar más de un rol.

2.3 PROCEDIMIENTOS DE BUENAS PRÁCTICAS

Definidos los roles de trabajo y su ubicación dentro de la propuesta de estructura organizativa general de un equipo de diseño, es conveniente que se pase a definir cuáles son las actividades que deberá realizar cada rol.

Es muy importante que se tenga en cuenta una vez más que cuando se emplea alguna metodología de desarrollo de software, ésta define en cierta medida qué actividades se realizan dentro del proceso de diseño y por quiénes son realizadas. Por esta razón la propuesta estará orientada en todo momento a apoyar y no a

sustituir el proceso. El modelo es lo suficientemente flexible como para poder adaptarlo a las condiciones reales de la organización que lo implemente.

Cada actividad se ejecuta por un rol determinado, en un momento determinado y manipula determinados artefactos de diseño. En este momento solo se hará referencia a *quién* realiza *qué* actividad. En el epígrafe 2.3.1 se estará abordando sobre el Expediente de Diseño y los artefactos que lo conforman.

Las principales actividades que se proponen realizará cada rol serán las siguientes:

- **Diseñador Principal:**
  - *Definición de los estándares de diseño*: esta actividad tiene una importancia significativa, ya que de ella depende que todo el equipo de diseñadores "hablen el mismo idioma". Dentro de sus principales aspectos estará la selección de algún lenguaje de modelado para la modelación gráfica del diseño (se propone UML). Esta actividad genera el ***Documento de Definición de Estándares del Proceso de Diseño***.

  - *Diseño Arquitectónico*: Esta es una actividad muy relacionada con el rol de Arquitecto del Sistema. En realidad pudiera interpretarse como tal. Se refiere a la modelación del sistema a nivel de componentes y sub-sistemas. El artefacto generado es el ***Documento de Diseño Arquitectónico***.

  - *Conformación del Expediente de Diseño*: Este es el principal artefacto que es generado por el proceso de diseño. Éste es constantemente actualizado y servirá como referencia principal de la organización para abordar temas relacionados con el proceso de diseño.

- *Reuniones de Planificación y Control*: Estas reuniones se realizarán con determinada sistematicidad definida por el equipo de diseño y respondiendo en todo momento a las necesidades de la organización.

- **Diseñador de la Lógica Funcional.**
  - *Diseño de Sub-Sistemas*: Definido el diseño arquitectónico del sistema, los detalles relacionados con la estructuración de cada subsistema, corresponde a este rol crearlos. Se deberá definir el comportamiento específico de cada subsistema en términos de colaboración entre los elementos de diseño contenidos y los subsistemas externos a través de las interfaces. El ***Modelo de Subsistemas*** será el artefacto generado en esta actividad.

  - *Diseño de Clases*: Corresponde a la modelación de las clases de diseño. En esta actividad se obtendrán los ***Diagramas de Clases del Diseño***.

  - *Diseño de la lógica funcional*: No es más que la modelación del diseño que responde a los requerimientos funcionales del cliente. Corresponde a la creación de ***Diagramas de Estados e Interacción***.

- **Diseñador del Modelo de Datos.**
  - *Análisis las Clases de Diseño*: Constituye el primer paso para la transformación del modelo de clases del diseño a un modelo de datos.

  - *Diseño del Modelo de Datos*: Es la construcción del ***Modelo de Datos*** en cuestión.
- **Diseñador de Mecanismos de Respaldo.**
  - *Diseño de Mecanismos de Recuperación de Errores*: Un importante momento en el funcionamiento del sistema, es la capacidad que tenga

éste de recuperarse de errores detectados. Estos mecanismos responderán a las necesidades del cliente en el sentido en que deberán ser hacia adelante o hacia atrás o partirán de alguna política de tolerancia a fallos. El artefacto asociado es el documento de ***Especificaciones del Modelo de Mecanismos de Recuperación de Errores y Tolerancia a Fallos***.

- *Diseño de Mecanismo de Manipulación de Excepciones*: se deberán definir el modelo de clases que responderá al tratamiento de excepciones. Para esta actividad se generará el ***Mecanismo de Excepciones***, el cual incluirá además una vista de las clases que definen dicho mecanismo.

- *Diseño de Modelo de Persistencia*: Esta actividad genera un importante artefacto para el desarrollo del sistema. El ***Esquema de Persistencia*** es el artefacto correspondiente a esta actividad y podrá generarse empleando algún motor de persistencia.

- *Definición de Mecanismos de mensajes a usuarios*: básicamente se refiere a la estandarización de la información que se le presentará al usuario.

- *Diseño de Mecanismos de seguridad*: La seguridad en un sistema va desde la manera en que éste se construya hasta la condición resultante de establecer y mantener medidas de protección que aseguren un estado de inviolabilidad ante actos o influencias hostiles.

- *Diseño de auditorías de procesos*: se refiere a la capacidad que deberá tener el sistema para registrar los eventos ocurridos o las acciones ejecutadas en el sistema. Esta actividad se realizará o no en

dependencia de los intereses de la organización y los requerimientos del usuario. En caso que se desarrollo, se generará el ***Especificaciones del Modelo de Auditorías de Procesos***.

- *Diseño de Mecanismos de Optimización de Recursos*: se refiere al manejo de la memoria; la manipulación del ciclo de los objetos, lo cual incluye la recolección de basura y reciclaje de objetos; los mecanismos de cacheado entre otros aspectos relacionados con el rendimiento del sistema.

- **Responsable de Integración.**
  - *Definición de estándares de integración*: estos estándares estarán orientados a la manera de integración del modelo de diseño de mecanismos de respaldo con la lógica funcional del sistema. Para el caso del empleo de *orientación a aspectos* para el diseño de los mecanismos de respaldo, se basará básicamente en la definición de los *puntos de enlaces*[16] donde se *tejerá*[17] el código correspondiente a la implementación de estos aspectos de respaldo del sistema.

- **Responsable de Mantenimiento.**
  - *Mantenimiento Sistemático del Diseño*: Esta actividad genera un artefacto que estará muy asociado a la gestión de cambio pero que se encargará de especificar el plan de mantenimiento, las razones de cambio, los cambios en sí y la descripción sobre la factibilidad de ejecutar el cambio: ***Especificaciones de Mantenimiento***.

---

[16] Es un término técnico de la Programación Orientada a Aspectos (AOP). Los puntos de enlace brindan la interfaz entre aspectos y componentes; son lugares dentro del código donde es posible agregar el comportamiento adicional que destaca a la
AOP. Dicho comportamiento adicional es especificado en los aspectos.

[17] Constituye otro término técnico de la AOP y se refiere a la manera de entrelazar el código de aspectos con el código de componentes por medio de los puntos de enlaces.

- **Revisor Técnico.**
  - *Revisiones Técnicas Formales*: Las revisiones técnicas devienen en importante mecanismo para la detección de errores en el diseño. Son empleadas en muchas metodologías y su efectividad es considerable partiendo de la idea de que un gran porciento de los errores del sistema es introducida durante la fase de diseño. Cada vez que se realice una revisión del diseño se generará un ***Registro de Revisión Técnica***, documento en el cual se reflejarán las principales observaciones realizadas en función de potenciales mejores, los errores encontrados –en caso de que existan- y sus posibles soluciones.

- **Analista de Reutilización.**
  - *Identificación de elementos de reutilización*: Básicamente consiste en realizar un estudio del estado del arte sobre diseños asociados a sistemas similares, con el objetivo de detectar posibles elementos reutilizables para el actual diseño en construcción. Esta actividad genera precisamente el ***Documento de Estudio del Estado del Arte de Reutilización***.

  - *Análisis de uso de Patrones de Diseño*: Análogamente, para el caso de los patrones el Analista de Reutilización deberá realizar un estudio asociado a los patrones que pueden ser empleados en el desarrollo del proceso y compatibles con la tecnología y los paradigmas que se estén empleando en la organización. Desde esta perspectiva se estará garantizando en cierta medida –entre otros aspectos- que el diseño actual sea lo más reutilizable posible. Para este caso se actualizará el ***Compendio de Patrones de Diseño*** que se genera en la actividad de

Análisis de Factibilidad Tecnológica, donde el principal objetivo de este artefacto es realizar una breve descripción sobre la base de la factibilidad, de los patrones objetos de estudio.

- **Responsable de Repositorio de Reutilización.**
  - *Administración del Repositorio de Reutilización*: este rol y el anterior trabajan en conjunto. El presente se encarga de administrar el ***Repositorio de Reutilización,*** precisamente construido partiendo del análisis realizado por el Analista de Reutilización. En esencia el repositorio no es más que el *nicho* donde se podrán encontrar aquellos diseños –o segmentos de diseños- o cualquier elemento de diseño reutilizable, que podrán ser reutilizados en la construcción del actual diseño.

- **Responsable de Refactorización.**
  - *Refactorización Continua*: esta actividad tiene estrecha relación con las revisiones técnicas y el mantenimiento del diseño. Se realizará de manera sistemática, controlada y buscando en todo la simplicidad en los cambios. En aras de optimizar el diseño, esta actividad deberá convertirse en *hábito* para este rol de trabajo y en todo momento deberá respetar las normas, principios y estándares sobre los cuales se encuentra trabajando el equipo de diseño.

  - *Documentar la actividad de refactorización*: El éxito de la refactorización depende en gran medida de la documentación de dicho procedimiento. La comunicación al resto del equipo sobre los cambios del diseño producidos por refactorizaciones, hará que éste tome posición en dichos cambios y sirva como evaluador principal de las nuevas modificaciones, además de que permitirá trabajar sobre las

actualizaciones y hacer evolucionar el diseño de manera organizada. El artefacto que se genera en esta actividad es el ***Documento de Refactorización***. Como es lógico, este documento será una fuente importante para el control y gestión de cambios en la organización.

- **Responsable de Calidad del Diseño.**
  - *Evaluación de Métricas de Diseño*: Dado a que el rol que deberá realizar esta actividad es miembro del equipo de calidad de la organización o proyecto, la misma por consiguiente formará parte de las actividades que desarrollará dicho equipo. Se seleccionarán las métricas de acuerdo a los atributos de diseño que se evaluarán y se generará un ***Documento de Evaluación de Métricas de Diseño*** en el cual aparecerán registradas las mediciones y se especificarán potenciales mejoras.

  - *Evaluación del Cumplimiento de Estándares de Diseño*: esta es una tarea complementaria cuyo objetivo es velar por el cumplimiento de los estándares de diseño definidos en el *Documento de Definición de Estándares de Diseño*, especificado por la Gerencia General de equipo.

- **Analista de Factibilidad Tecnológica.**
  - *Análisis de Factibilidad Tecnológica*: el análisis de factibilidad tecnológica estará orientada a los sistemas que se emplearán para la automatización del proceso de diseño, la generación de código y el modelaje gráfico. Básicamente se realizará un estudio del estado del arte de las diferentes tecnologías, para sobre la base de los principios de la organización, en cuanto a la plataforma de desarrollo, el empleo o no de sistemas propietarios o sistemas bajo licencias libres, definir la

plataforma tecnológica sobre la cual se estará desarrollando el proceso de diseño en cuestión. El artefacto que se genera es el ***Documento de Análisis de Factibilidad Tecnológica***.

Estas son las principales actividades que se proponen se realicen en el proceso de diseño, partiendo de la propuesta de estructura organizacional en departamentos y la definición de los respectivos roles de trabajo que ejecutarán las antes mencionadas actividades.

2.3.1 EXPEDIENTE DE DISEÑO

El *Expediente de Diseño* es un artefacto por sí solo, de hecho, es el principal artefacto que genera el proceso de diseño. Este es un elemento que se propone sea empleado como principal mecanismo para presentar el avance y la forma de trabajo desde la perspectiva del diseño.

En sentido general podría decirse que el expediente de diseño no es más que un archivo que comprende la construcción de un documento denominado ***Informe General del Proceso de Diseño***, en el cual se recoge información relacionada con los elementos estructurales de conformación del equipo de diseño, el modo de trabajo, la definición de estándares y una visión general de los artefactos más importantes generados por los departamentos del proceso de diseño. La otra parte del expediente de diseño es la conformación íntegra de todos los artefactos del proceso sobre su formato original. Es decir, el *archivo,* como elemento aglutinador, define un conjunto de secciones análogas a la definición estructural del proceso y en las que se incluye un espacio donde quedarán almacenados los artefactos correspondientes a cada departamento.

Para que se comprenda con más claridad, la estructura general que se propone del Expediente de Diseño es la siguiente:

- Documentos Generales.
    - Informe General del Proceso de Diseño.
    - Actas de Reuniones de Control.

- Artefactos:
    - Gerencia General del Proceso de Diseño (GGPD):
        - Documento de Definición de Estándares del Proceso de Diseño.
        - Documento de Diseño Arquitectónico.

    - Gestión y Evaluación de Calidad del Diseño (GECD):
        - Documento de Evaluación de Métricas de Diseño.

    - Análisis y Factibilidad Tecnológica (AFT):
        - Documento de Análisis de Factibilidad Tecnológica.

    - Reutilización y Refactorización (RR):
        - Documento de Estudio del Estado del Arte de Reutilización.
        - Compendio de Patrones de Diseño.
        - Repositorio de Reutilización.
        - Documento de Refactorización.

    - Diseño de la Lógica Funcional del Sistema (DLFS):
        - Modelo de Subsistemas.
        - Diagramas de Clases del Diseño.
        - Diagramas de Estados e Interacción:
            - Diagrama de Secuencia.
            - Diagrama de Colaboración.
            - Diagrama de Estados.

- Diseño de Mecanismos de Respaldo (DMR):
  - Mecanismo de Excepciones.
  - Esquema de Persistencia.
  - Especificaciones del Modelo de Auditorías de Procesos.
  - Especificaciones del Modelo de Mecanismos de Recuperación de Errores y Tolerancia a Fallos.

- Mantenimiento de Diseño (MD):
  - Especificaciones de Mantenimiento.
  - Registro de Revisiones Técnicas.

El desarrollo o no de algunos de estos artefactos estará en dependencia de los objetivos de la organización y de su modo de trabajo. La propuesta realizada responde al cumplimiento del objetivo general que persigue el presente trabajo, de brindar una propuesta de guía metodológica que sirva como soporte para el desarrollo del proceso de diseño de software.

Ninguna de las metodologías de desarrollo de software existentes hoy en el mundo hace una propuesta de conformación de un *Expediente de Diseño* como artefacto del proceso. Esto quizás pueda parecer una consecuencia de no ser necesario dicho artefacto para la organización o proyecto; o para simplificar aún más la dinámica del proceso de diseño. Lo real del asunto es que pueda deberse a las dos razones expuestas, sin embargo, desde la perspectiva de *necesidad* y a favor del empleo de esta estructura, se ganaría significativamente en ***organización***, ***estandarización del proceso, rigurosidad*** y a la vez ***facilidad en el control*** además de servir como ***soporte*** para desarrollar el proceso de diseño en sí.

Desde otro ángulo y relacionado a la simplificación del proceso como segunda razón, el modelo propuesto está caracterizado por una ***flexibilidad*** capaz de hacerlo lo más fácilmente adaptable en el ámbito de cualquier organización. La rigurosidad a la que

se hacía referencia, no debe constituir un elemento que obstaculice la *agilidad* con la que se desarrolle la dinámica del proceso, ni complejizar el desempeño del equipo de desarrollo. Por esta razón el empleo o no del modelo queda a decisión de la organización y en dependencia de los objetivos de la misma.

2.3.1.1 ARTEFACTOS

Cada uno de los artefactos que define la propuesta del presente trabajo tiene un objetivo en específico y tributa determinada información al proceso para que éste pueda desarrollarse lo más efectivamente posible y con calidad.

Es preciso se aclare que el emplear estos artefactos no garantizará la calidad del diseño como proceso ni como producto sino que servirá como soporte para que el proyecto gane en control y organización, dos elementos que favorecen en gran medida a la calidad.

Cada artefacto es el resultado de un subproceso del diseño, lo que indica que para lograr un incremento de la calidad en el proceso deberá realizarse cada tarea de manera correcta, respetando los principios del diseño y el análisis sobre el empleo de la tecnología, el uso de patrones de diseño y los principios de organización y estructuración de un equipo de diseño; correspondencia con los requerimientos de los clientes y el dominio del problema; una adecuada evaluación de métricas y modelos de calidad; además de construir cada artefacto de manera adecuada, legible y ajustado a los estándares definidos por la organización, entre otros aspectos que fueron mencionados en epígrafes anteriores.

La descripción general de cada artefacto es la siguiente:

- **Documento de Definición de Estándares del Proceso de Diseño**: es un documento que es desarrollado por el *Diseñador Principal* y en colaboración con el departamento de calidad del proceso, es evaluada su factibilidad y

controlado su cumplimiento. El objetivo principal del mismo es definir los estándares desde los que deberá partir el proceso. Dichos estándares recogerán aspectos relacionados con:

- Métodos de trabajo.
- Estilos de diseño.
- Plantillas para el desarrollo de artefactos.
- Modelos de Calidad.
- Métricas de Diseño.
- Principios organizativos.

El documento parte de la definición de una plantilla que organiza la estructuración del contenido y define los elementos de autoría, control de versiones y revisiones y el modo de redacción (*ver ANEXO 10*).

Una vez que es creado deberá ser presentado a la gerencia general del proyecto u organización para su aprobación, y posteriormente gestionado por el departamento de calidad el cual deberá divulgarlo al resto del equipo de diseño para hacer cumplir lo establecido en el mismo.

- **Documento de Diseño Arquitectónico:** este es un documento que no deberá causar polémica dentro del equipo de desarrollo por su estrecha relación con la actividad de definición de la arquitectura del sistema, especificada por algunas metodologías como parte de las actividades del *arquitecto* y no como resultado del proceso de diseño. Metodologías como RUP definen su propio documento de arquitectura con sus especificidades concretas, como es el caso de las 4+1 vistas de RUP.

  En todo momento se parte del criterio de que el diseño arquitectónico es una etapa del proceso de diseño, por lo que de acuerdo a la metodología que emplee la organización para desarrollar el sistema, se definirá o no un rol de

arquitecto, o se le será asignada ésta responsabilidad al *Diseñador Principal* como rol propuesto por el presente modelo.

Sea cual sea la determinación de la gerencia del proyecto u organización, el objetivo de este documento es brindar una vista arquitectónica del sistema en construcción, desde la perspectiva de estructuración de los datos y los componentes del programa que se desea construir, en subsistemas, módulos y/o paquetes; así como las características de cada componente y la forma en que éstos interactúan (*ver ANEXO 11*).

- **Documento de Evaluación de Métricas de Diseño:** el resultado de evaluar métricas de calidad orientadas al diseño de software será plasmado en el presente documento, cuyo objetivo principal es servir de fuente de consulta para el equipo de diseño, sobre los atributos de calidad del proceso y sus potenciales mejoras.

  El mismo se encargará de especificar la familia de métricas empleadas y el resultado de los cálculos realizados. En caso que se emplee algún sistema que automatice el proceso de evaluación de métricas de calidad –como el modelo RARER-, el documento podrá servir solo como *informe de resultados generales (ver ANEXO 12)*.

  Documento de Análisis de Factibilidad Tecnológica: este es un importante documento para el proceso de automatización del diseño. El objetivo del mismo es realizar un estudio sobre las herramientas que permitan automatizar el proceso de diseño, sobre la base de la tecnología y los principios definidos por la organización o proyecto, relacionados con patentes, sistemas operativos, situación del hardware, entre otros aspectos. Por esta razón, el documento se encargará de analizar la factibilidad de emplear determinado sistema o lenguaje de modelado.

Sobre la selección de alguna herramienta CASE para el modelado, preferentemente en UML, se deberá centrar el análisis en los siguientes criterios de selección:

- Facilidad de modelado visual de artefactos UML.
  - Validación de sintaxis UML.
  - Presentación de modelos.
  - Facilidades de dibujo.

- Organización del Repositorio de artefactos UML.
  - Establecer jerarquías de paquetes de artefactos
  - Control de accesos y versiones para múltiples usuarios
- Versiones de UML que soporta.

- Extensiones de Modelado con UML
  - Modelado de procesos de negocio empleando BPMN.
  - Modelado de datos.
  - Modelado de aplicaciones Web.
  - Modelado de sistemas de tiempo real.

- Procesos Ingenieriles
  - Empleo de *Forward Engineering* desde modelos de objetos a modelos de datos y esquemas de persistencia.
  - Capacidad para la realización de *Ingeniería Inversa (Reverse Engineering).*

- Exportación e importación de modelos.
- Generación de Documentación.
- Integración con herramientas de trazabilidad de requerimientos para la realización de *back tracking.*

- Integración con herramientas de control de versiones.
- Soporte de implementación y generación de código.
- Hipervinculación de artefactos de modelado con documentos de especificación.
- Integración de patrones de diseño.
- Facilidad de uso.
- Compatibilidad con otros sistemas que se empleen en la organización para automatizar otros procesos de desarrollo.

Estos son algunos de los elementos que podrían tenerse en cuenta a la hora de seleccionar alguna herramienta de modelado para automatizar el proceso de diseño. Quedará por parte de la organización asumir el nivel de profundidad con que se realizará el análisis de estos aspectos (*ver ANEXO 13*).

- **Documento de Estudio del Estado del Arte de Reutilización:** el objetivo de este artefacto es realizar un estudio sobre la posibilidad de reutilización de diseños externos a la organización o internos en ella. Esto se refiere a empleo de frameworks, secciones de diseño de otros sistemas y utilización de patrones de diseño. Su alcance será hasta la posibilidad de reutilizar código para minimizar el tiempo y los costes de desarrollo. Otro aspecto a tratar en el mismo será la evaluación de la compatibilidad de lo que se reutilice respecto a los principios y estándares definidos por el equipo de diseño y la organización en sentido general. Por esta razón, se incluirá el análisis sobre el ajuste de los elementos reutilizables a dichos principios y estándares (ver ANEXO 14).

- **Compendio de Patrones de Diseño:** el compendio de patrones es un documento que definirá el paquete de patrones de diseño que podrán emplearse en la construcción del diseño, sobre la base de la tecnología y los paradigmas que fungen como pilares para el desarrollo del sistema en sentido

general. El análisis no se centrará en la factibilidad de uso de determinado patrón en específico sino en el paquete en sentido general (*ver ANEXO 15*).

- **Repositorio de Reutilización:** este es un importante artefacto dado a que constituye el nicho de reutilización más valioso para el proceso de diseño. Básicamente será un *archivo físico*[18] donde podrán consultarse posibles elementos a reutilizar. Es un artefacto que está muy relacionado con el *Documento de Estudio del Estado del Arte de Reutilización* dado a que es consecuencia de él. Será administrado por el *Responsable de Repositorio de Reutilización* el cual a su vez se encargará de divulgar a todo el equipo de diseñadores los cambios y actualizaciones que en él se realicen así como también de los elementos que lo compongan y que puedan ser reutilizados en determinado momento del proceso de diseño.

- **Documento de Refactorización:** este documento es una especie de control de cambios del diseño, pero que además incluye las razones por las cuales se ejecutaron dichos cambios. La refactorización en el proceso de diseño se realizará de manera continua, por lo que el documento estará en constante actualización. Una parte importante de este proceso, tal y como se mencionaba en epígrafes anteriores, es la divulgación de dicho cambio al resto del equipo de diseñadores (*ver ANEXO 16*).

- **Modelo de Subsistemas:** constituye una vista del sistema en subsistemas de diseño, mostrando la forma en que éstos interactúan entre sí y otros sistemas externos.

- **Diagramas de Clases del Diseño:** éste es uno de los más importantes diagramas estáticos. Su objetivo es mostrar cómo están diseñadas las clases

[18] Entiéndase como el lugar donde se encuentren todos los elementos reutilizables del proceso de diseño.

del sistema y sus interacciones. Se parte del mismo principio de diagrama de clases de diseño que brinda UML y emplean otras muchas metodologías de desarrollo como RUP, XP, entre otras. Sería absurdo hablar de la construcción de éste tipo de diagramas de un modo manual, cuando existen innumerables herramientas que automatizan este proceso. El empleo de una determinada herramienta para construir los diagramas del proceso de diseño, será un resultado del análisis de factibilidad tecnológica realizado en etapas anteriores.

- **Diagramas de Estados e Interacción:** la definición de este tipo de diagrama parte de la que brinda UML propiamente. Los mismos sirven para ilustrar el modo en el que los objetos interaccionan por medio de mensajes y se dividen en dos tipos de diagramas: los de secuencia y los de colaboración.
    - Diagrama de Colaboración: ilustra las interacciones entre objetos en un formato de grafo o red, en el cual los objetos se pueden colocar en cualquier lugar del diagrama (LARMAN, 2003) y sirve para entender el orden en que se envían los mensajes.
    - Diagrama de Secuencia: por otra parte, los diagramas de secuencias ilustran las interacciones en un tipo de formato con el aspecto de una valla, en el que cada objeto nuevo se añade a la derecha (LARMAN, 2003). Tiene un objetivo similar al diagrama de colaboración.

  Por otro lado están los Diagramas de Estados, los que constituyen una manera para caracterizar un cambio en un sistema, es decir, que los objetos que lo componen modificaron su *estado* como respuesta a los sucesos y al tiempo (SCHMULLER, 2000).

- **Mecanismos de Excepciones:** en realidad los mecanismos de excepciones son más que un simple modelo de clases basado en aserciones u orientado a

aspectos[19]. Para este artefacto se deberán tener diferentes consideraciones relacionadas con las políticas de tratamiento de excepciones, estándares de mensajes, además de la infraestructura de clases que soporta esa lógica de tratamiento (*ver ANEXO 17*).

- **Esquema de Persistencia:** en principios no es más que un modelo conceptual de las clases que definen los datos que persistirán sobre algún formato.

- **Especificaciones del Modelo de Auditorías de Procesos:** este es un artefacto opcional dentro del proceso de desarrollo de un sistema, de hecho, no es el único artefacto opcional pero dado a que un sistema puede o no tener algún mecanismo de auditorías de procesos lo hace un artefacto potencialmente opcional. Sin embargo, la definición de procedimientos que auditen todo sobre lo que en el sistema los usuarios realicen, le imprime al software un mayor nivel de fiabilidad además de ser una buena práctica para el seguimiento de la seguridad y soporte para el mantenimiento de la aplicación. Básicamente en este artefacto se definirán las políticas de auditorías, se describirán los principales procesos que serán auditados en el sistema así como también se especificará sobre algún framework empleado para la implementación de dicho mecanismo, los estándares a seguir para los reportes de auditorías y la forma en que serán gestionados (ver ANEXO 18).

- **Especificaciones del Modelo de Mecanismos de Recuperación de Errores y Tolerancia a Fallos:** este es un artefacto complementario. Si bien es cierto que el tratamiento de excepciones es un mecanismo de recuperación de errores *hacia adelante,* no es la única manera de gestionar los errores en un sistema ni el único mecanismo de tratamiento de errores. Por otro lado está el nivel de *tolerancia* que defina el sistema para "admitir" en determinado grado,

[19] Referente a la Programación Orientada a Aspectos.

la ocurrencia de un error. Este tema, al igual que el tratado en el artefacto anterior, están muy asociados al tema de *fiabilidad y robustez* de un sistema, por lo que el desarrollarlo o no estará en dependencia de los objetivos de la organización y los requerimiento especificados por el cliente.

Este es un documento en el que deberán plasmarse las políticas de tratamiento de errores y la tolerancia o no a fallas del sistema. Si se decide realizar este artefacto, no será preciso construir el artefacto *Mecanismos de Excepciones* ya que se encuentra implícito en el presente (*ver ANEXO 19*).

- **Especificaciones de Mantenimiento:** este es un documento en el cual se definirá la planificación del mantenimiento del diseño, se registrarán los cambios realizados y las causas que los provocaron. Es un artefacto que será manipulado por gran parte del equipo del diseño y estará constantemente bajo actualizaciones sistemáticas, determinadas por la frecuencia con que se realice el mantenimiento (*ver ANEXO 20*).

- **Registro de Revisión Técnica:** Este es un documento de anotaciones generado como consecuencia de la realización de una revisión técnica al diseño. Su principal objetivo es realizar un registro de los resultados de haber realizado una revisión al diseño o a una sección de éste. Es un artefacto que tributa información al proceso de mantenimiento y sirve de soporte para llevar un control de registro de errores del diseño (*ver ANEXO 21*).

- **Expediente de Diseño:** Como se mencionaba en ocasiones anteriores, éste es el principal artefacto propuesto para ser generado por el proceso de diseño. Su principal objetivo es brindar a la organización toda la información necesaria sobre el proceso de diseño. Está seccionado en dos partes: la definición íntegra de todos los artefactos del proceso (*ver epígrafe 2.3.1*) y el ***Informe General del Proceso de Diseño.***

- **Informe General del Proceso de Diseño:** este es un documento que resume los principales elementos del proceso de diseño. En sentido general, sirve para poder *navegar* por todo el proceso de una manera rápida y precisa. Es sistemáticamente actualizado y constituye una herramienta para poder mostrar al resto de la organización el avance del proceso de diseño (*ver ANEXO 22*).

Es preciso se aclare una vez más que el empleo o no de estos artefactos quedará a decisión del equipo de diseño y más específicamente de la organización. Es importante se comprenda que la propuesta de estos artefactos es con el objetivo de brindar un conjunto de elementos en los cuales puedan apoyarse los miembros del equipo de diseño para manejar información del proceso y tener un soporte estándar para poder divulgarla internamente. Sumarle a esto que en todo momento deberá evaluarse la factibilidad de empleo o no de algún artefacto en función de las necesidades y objetivos de la organización. Si el equipo considera que existen artefactos que retrasarían el proceso o lo harían más complejo, pues debieran limitarse a emplear solo aquellos que brinden un verdadero aporte.

2.4 CRITERIOS DE ESPECIALISTAS

Un importante momento dentro de la propuesta de solución que en el presente trabajo de diploma se presenta, es el de validar los elementos y criterios de buenas prácticas para evaluar la factibilidad de aplicarlos en un entorno real.

La validación es una actividad científica y se realiza por algún método definido para tal objetivo.

Por el momento sólo se aplicó, como antesala a dicha validación, una encuesta a especialistas como un mecanismo de obtención de criterios de expertos. En trabajos futuros se desarrollará un proceso de validación en dos etapas y empleando dos métodos: Método de Expertos y el Método de Experimentación (*vea epígrafe 2.5*).

La encuesta tuvo como objetivo específico recopilar información sobre criterios de especialistas para evaluar la factibilidad de la propuesta de solución; la cual servirá para apoyar el proceso de validación de la solución propuesta y su alcance estará correspondido con la recopilación de criterios sobre cuestiones asociadas al proceso de diseño en cuestión. Fue aplicada a un número reducido de especialistas y estará complementada por otros mecanismos de validación del modelo propuesto, que serán aplicados en trabajos posteriores.

Se seleccionó como muestra a los especialistas del proyecto de PDVSA (*Petróleos de Venezuela S.A.*) encargados de desarrollar la interfaz web del sistema *SENTAI*[20] empleado para la gestión de los procesos de ventas, contabilidad, inventario, etc., asociados al despliegue de puntos de ventas y almacenaje de alimentos en todo el país de Venezuela y un especialista de la empresa colombiana *Outsorcing en Desarrollos Informáticos Ltda. (ODI).*

La encuesta estuvo basada en los siguientes elementos (*ver ANEXO 23*):

- Conocimiento sobre alguna ***metodología*** para realizar el proceso de diseño en el desarrollo de algún sistema de software.

- Conocimiento de algún ***procedimiento o método*** que permita establecer pautas o principios para la realización del proceso de diseño dentro de una organización o proyecto de desarrollo de software

- Conocimiento sobre algún ***método*** para organizar la estructuración de un equipo de diseño y definir y asignar roles entre los miembros de éste.

- Consideraciones sobre la importancia de algún modelo o procedimiento de buenas prácticas de diseño que apoye la toma de decisiones en el proceso y contribuya al mejoramiento de los atributos de calidad.

[20] Sistema de gestión empresarial empleado para el control de los procesos de ventas, inventario, contabilidad, etc., de la distribución de alimentos a escala nacional en Venezuela.

- Valoración sobre la factibilidad o no de aplicar la propuesta de solución en el desarrollo de software en un entorno real.

De acuerdo con los resultados de la encuesta se pudo llegar a las siguientes observaciones:

1. Ninguno de los especialistas encuestados conocían de *metodología* alguna para la realización del proceso de diseño específicamente. Muchos empleaban los elementos que brindaba la metodología de desarrollo que en el momento se utilizaba para construir el software, como RUP y RAD.

2. Ninguno de los especialistas encuestados conocían de algún *procedimiento o método* que permita establecer principios o pautas para desarrollar el proceso de diseño.

3. De la misma manera en que planteaban que desconocían sobre la existencia de algún *método* que permitiera definir la estructura organizacional de un equipo de diseño y la asignación de roles a sus respectivos miembros. Solo un especialista realizaba este proceso de estructuración basándose en lo empírico.

4. Sobre el conocimiento de la existencia o no de roles para el proceso de diseño, coincidieron en que sólo conocían del rol de *diseñador de software.*

5. Estuvieron de acuerdo en que para el mejoramiento de los atributos de calidad del diseño, es importante el empleo de algún modelo o procedimiento de buenas prácticas que contribuya a la toma de decisiones en el proceso.

6. En tanto calificaron la propuesta de solución a partir de los siguientes criterios:

- Necesaria pero no suficiente para organizar y desarrollar el proceso de diseño.
- Importante para organizar y desarrollar el proceso de diseño.
- Recomendable para el proceso de diseño por las prácticas que propone.

7. En sentido general, todos estuvieron de acuerdo en que la aplicación de la guía de buenas prácticas propuesta podría ser factible en el proceso de desarrollo de software.

Otros planteamientos estuvieron dirigidos a la necesidad de validar el modelo propuesto para evaluar cuán factible y efectivo es para aplicar en una empresa o proyecto de desarrollo real.

En el caso de la definición y asignación de roles, el especialista de *Outsorcing en Desarrollos Informáticos Ltda. (ODI)* definía para el proceso de diseño tres roles principales:

- *Analista*: el cual realiza el levantamiento de las necesidades funcionales del Cliente. Y diseña el modelo a desarrollar, utilizando alguna herramienta UML.

- *Programador*: Se encarga del código y su desarrollo. Además de la corrección de errores y la puesta en marcha.

- *Control de Calidad:* este rol era desempeñado por el líder usuario del software que se desarrolla.

En la práctica se ha podido constatar lo fácil que se disuelven las responsabilidades y roles asignados a los miembros del equipo. Lo visible que se hace el tema de la competitividad entre los mismos, lo que conlleva a una falta de comunicación.

Innumerables proyectos han fracasado por esto último. Además, la falta de comunicación y contactos frecuentes con el cliente, que implica, en la posterioridad del desarrollo, un análisis de abajo hacia arriba, lo que impacta en las definiciones del diseño, en el tiempo de entrega y pos supuesto, en los costes.

La presente propuesta se caracteriza por su fácil entendimiento y su facilidad de navegabilidad, tal y como fue reconocido por los especialistas en el estudio realizado.

2.5 TRABAJOS FUTUROS

Los trabajos futuros estarán orientados a la realización de dos actividades fundamentales: la validación de la propuesta de guía metodológica de buenas prácticas y en segundo lugar, el refinamiento de la misma de acuerdo a los resultados obtenidos de los procesos de validación.

Se propone que la validación se desarrolle en dos momentos y mediante dos métodos. Un primer momento será la validación mediante el *Método de Expertos* para evaluar la factibilidad de realizar o no una validación mediante el *Método de Experimentación* dado a que este último requiere de mayor esfuerzo, se caracteriza por una mayor complejidad y los recursos implicados suelen ser costosos, además de que dicho costo puede ser aún mayor si los resultados no son los esperados llegándose a obtener pérdidas en lugar de ganancias.

El refinamiento de la guía metodológica es un importante paso para lograr ajustar los elementos y principios propuestos a las reales condiciones en que operan las empresas, proyectos y organizaciones para el desarrollo de software. Es un proceso que permite el enriquecimiento y una mejor argumentación de la propuesta de solución, además que imprime un carácter continuo al proceso y por ende posibilita la evolución de los mecanismos y procedimientos propuestos.

CONCLUSIONES GENERALES

Concluir no sería posible hasta tanto se pudiese validar el modelo propuesto para evaluar cuán efectivo es de aplicar en el desarrollo de algún software en un contexto real. Sin embargo, de acuerdo con el objetivo general que anima la dinámica del presente trabajo; el problema propuesto y partiendo de que existe una necesidad real de introducir elementos que estandaricen el proceso de diseño y sirvan de soporte para la toma de decisiones en una organización o proyecto de desarrollo, se puede decir que la presente guía metodológica de buenas prácticas orientada a la calidad del diseño, constituye un modelo que sirve de apoyo para poder desarrollar el proceso de diseño de software y podría resultar muy efectiva su aplicación, partiendo de las consideraciones generales realizadas por los especialistas encuestados.

BIBLIOGRAFÍA

1. **AMELLER, David y FRANCH, Xavier.** *Asignación de Tratamientos a Responsabilidades en el contexto del Diseño Arquitectónico Dirigido por Modelos.* Departamento de Llenguajes y Sistemas Informáticos, Universidad Politècnica de Cataluña. Barcelona : s.n. pág. 10.

2. **AMENDOLA, L. 2003.** *La Confiabilidad Desde el Diseño. Proyectos de Mantenimiento.* Departamento de Proyectos de Ingeniería, PMM Institute for Learning. Universidad Politécnica de Valencia. España : Asociación Española de Mantenimiento, 2003. pág. 10, Artículo.

3. **BARBACCI, Mario. 2005.** IEEE Argentina. *IEEE.* [En línea] IEEE, 22 de Julio de 2005. [Citado el: 9 de Febrero de 2008.] http://www.ieee.org.ar/noticiasdetalle.asp?IDNoticia=115. CS-08.

4. **BOHORQUEZ, José Yván. 2008.** Factorías de Software. Implementación de una Factoría en la empresa E4GS. [entrev.] Jorge Luis VALDÉS GONZÁLEZ. *Factorías de Software.* Los Cortijos, 29 de Abril de 2008.

5. **BOOCH, Grady. 1998.** *Oject-Oriented Analysis and Design whith applications.* Segunda Edición. Santa Clara : Addison-Wesley, 1998. 0-8053-5340-2.

6. **BOOCH, GRADY y RUMBAUGH, JAMES. 1995.** *Unified Method for Object-Oriented Development.* s.l. : Rational Software Corporation, 1995. Overview.

7. **CABRERO, Daniel, GARZÁS, Javier y PIATTINI, Mario. 2008.** *Técnica de Mejora del Mantenimiento Software Basada en Valor.* España : s.n., 2008.

8. **CANÓS, José H., LETELIER, Patricio y PENADÉS, Carmen. 2003.** *Metodologías Ágiles en el Desarrollo de Software.* Universidad Politécnica de Valencia (DSIC). España : Ingeniería del Software y Sistemas de Información (ISSI), 2003. Informe. Artículo perteneciente al Informe del Taller Internacional sobre las Metodologías Ágiles en el Desarrollo de Software..

9. **CARLES AMBROJO, Joan. 2005.** Las fábricas de 'software' buscan especialización y costes laborales más bajos. *El País.* [En línea] 12 de Mayo de 2005. [Citado el: 4 de Abril de 2008.] http://www.elpais.com/articulo/tecnologia/fabricas/software/buscan/especializacion/costes/laborales/bajos/elpcibtec/20050512elpcibtec_3/Tes.

10. **Diario El País. 2008.** Ciberpaís. *El País.* [En línea] 4 de Abril de 2008. [Citado el: 4 de Abril de 2008.] http://www.elpais.com.

11. **Empresa de Software © DAEDALUS. 2007.** DAEDALUS. [En línea] Data, Decisions and Language, S. A., 2007. [Citado el: 8 de Enero de 2008.] http://www.daedalus.es/AreasISDiseno-E.php.

12. **FENTON, Norman E. y NEIL, Martin. 2005.** *Software Metrics: Roadmap.* London : s.n., 2005.

13. **FENTON, Norman E. y PFLEEGER, Shari Lawrence. 1997.** *Software metrics : a rigorous approach.* 2nd. London : Boston : PWS Pub., 1997. 0534954251.

14. **GALVÁN, Blas J., y otros. 2005.** *Técnicas de análisis de fiabilidad y métodos evolutivos de optimización global aplicados al diseño de software.* Instituto de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería, Universidad de

Las Palmas de Gran Canaria. Islas Canarias : s.n., 2005. pág. 17, Artículo de Investigación.

15. **GAMMA, Erich, y otros. 1998.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison Wesley Longman, 1998. 0-201-63498-8.

16. **GARCÍA PEÑALVO, Francisco José, y otros. 2004.** *Modelo de Cualificación de Assets del repositorio GIRO.* Valladolid : s.n., 2004.

17. **GARCIA, Joaquín. 2006.** Gestión de proyectos con SCRUM. *IngenieroSoftware.* [En línea] 4 de Septiembre de 2006. [Citado el: 18 de Marzo de 2008.] http://www.ingenierosoftware.com/equipos/scrum.php.

18. **GONZÁLEZ ARECHAVALA, Yolanda y GARCÍA, Fernando de Cuadra. 2001.** *Calidad del Software.* Instituto de Investigación, Escuela Técnica Superior de Ingeniería (ICAI). s.l. : País Vasco, 2001. pág. 54.

19. **GRACIA, Joaquín. 2004.** Podredumbre del software. *IngenieroSoftware.* [En línea] 22 de Agosto de 2004. [Citado el: 18 de Marzo de 2008.] http://www.ingenierosoftware.com/analisisydiseno/podredumbre.php.

20. **GREENFIELD, Jack y SHORT, Keith. 2003.** *Software Factories. Assembling Applications with Patterns, Models,Frameworks and Tools.* Microsoft. California, USA : s.n., 2003. 13.

21. **HUMPHREY, WATTS S. 2001.** *Introducción al Proceso de Software Personal.* [ed.] Andrés Otero. Primera Edición. Madrid : PEARSON EDUCATION S.A., 2001. pág. 328. Edición en Español. 84-7829-052-4.

22. **IEEE. 2004.** *Guide to the Software Engineering Body of Knowledge.* s.l. : IEEE Computer Society, 2004. 0-7695-2330-7.

23. **—. 1990.** *IEEE Standard Glossary of Software Engineering Terminology.* Institute of Electrical and Electronics Engineers (IEEE). s.l. : IEEE, 1990. pág. 84. 155937067X.

24. **—. 2006.** IEEE, The world's leading professional association. [En línea] IEEE, 2006. [Citado el: 3 de Febrero de 2008.]

25. **I-Sol S.A. Intelligent Solutions. 2008.** *Software Factory. Inserción de I-Sol S.A. como empresa argentina, en el mercado mundial.* Capital Federal de Argentina : s.n., 2008.

26. **Iván, Ing. 2008.** Factorías de Software. Implementación de una Factoría en la empresa E4GS. [entrev.] Jorge Luis VALDÉS GONZÁLEZ. *Factorías de Software.* Los Cortijos, 29 de Abril de 2008.

27. **IZQUIERDO HERRERA, Raykenler y LAZO OCHOA, René. 2007.** *El modelo de diseño del sistema HyperWeb. Módulos de Tratamiento Farmacológico y Configuración.* Ciudad de la Habana : s.n., 2007.

28. **JACOBSON, IVAR, BOOCH, GRADY y RUMBAUGH, JAMES. 2000.** *El Proceso Unificado de Desarrollo de Software.* [ed.] Andrés Otero. [trad.] Salvador Sánches. Madrid : PEARSON EDUCACIÓN S.A., 2000. pág. 464. Edición en Español. 0-201-57169-2.

29.—. **2000.** *El Proceso Unificado de Desarrollo de Software.* [trad.] Salvador Sánchez, y otros. 1. Madrid : PEARSON EDUCACIÓN S.A., 2000. pág. 464. Edición en Español. 84-7829-036-2.

30. **JAUHAR, ALI y TANAKA, JIRO. 1998.** *An Object Oriented Approach to Generate Executable Code from the OMT-base Dynamic Model.* Instituto de Siencias Informáticas y Electrónica, Unversidad de Tsukuba. Japón : s.n., 1998. Informe de Investigación.

31. **JETTER, Andreas. 2006.** *Assessing Software Quality Attributes. With Source Code Metrics.* Department of Informatics, University of Zurich. Zurich : Software Evolution & Architecture Lab (SEAL), 2006. pág. 56, Tesis de Diploma.

32. **KNIBERG, Henrik. 2007.** *Una historia de guerra ágil. SCRUM y XP desde las trincheras.Cómo hacemos SCRUM.* [ed.] Diana PLESA. [trad.] Ángel MEDINILLA. EE.UU. : Enterprise Software Development Series (InfoQ), 2007. Enterprise Software Development Series. 978-1-4303-2264-1.

33. **LARMAN, Craig. 2004.** *Applying UML And Patterns.* s.l. : Prentice Hall, 2004. 9780131489066.

34.—. **2003.** *UML y Patrones. Introducción al análisis y diseño orientado a objetos.* Segunda Edición. México : PEARSON EDUCACIÓN , 2003. pág. 624. Vol. 1, Edición en Español. 8420534382 .

35. **LÓPEZ, Oscar, LAGUNA, Miguel Ángel y MARQUÉS, José Manuel. 2004.** *Reutilización del Software a partir de Requisitos Funcionales en el Modelo de Mecano: Comparación de Escenarios.* Universidad de Valladolid. 2004. pág. 12.

36. **MARÍN, Beatriz, CONDORY-FERNÁNDEZ, Nelly y PASTOR, Oscar. 2007.** Calidad de Modelos Conceptuales. Un análisis multidimensional de modelos cuantitativos basados en la ISO 9126. [ed.] Dr. D. José C. VERDÚN y Dr. D. José Antonio GUTIERREZ. *Revista de Procesos y Métricas de las Tecnologías de la Información.* Edición Especial, Octubre de 2007, 4, pág. 92. Número Especial con motivo de la VIII Conferencia Anual de la Asociación Española de Métricas de Sistemas Informáticos. III Encuentro Internacional ISBSG-AEMES.

37. **MEYER, Bertrand. 1997.** *Object-Oriented Software Construction.* ISE, Santa Barbara : Prentice Hall, 1997. 84-8322-040-7.

38. **MILLER, Robert y TRIPATHI, Anand. 2003.** *Primitives and Mechanisms of the Guardian Model for Exception Handling in Distributed Systems.* Computer Science Department, University of Minnesota. Minneapolis : s.n., 2003. Artículo de Conferencia. Conferencia Internacional ECOOP'2003 (Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms).

39. **MONASOR, Miguel Jiménez. 2008.** *Calidad y Medición de Sistemas de Información. Calidad de Proceso. Una orientación hacia el Desarrollo Distribuido de Software.* ALARCOS. Ciudad Real : s.n., 2008. pág. 21.

40. **MUÑOZ, Javier y PELECHANO, Vicente. 2007.** *MDA vs Factorías de Software.* Dept. de Sistemas Informáticos y Computadores, Universidad Politécnica de Valencia. Valencia : s.n., 2007. pág. 10, Artículo.

41. **Object Management Group (OMG). 2003.** *MDA Guide Version 1.0.1.* s.l. : OMG, 2003. pág. 62.

42. **OLMEDILLA ARREGUI, Juan José. 2005.** *Revisión Sistemática de Métricas de Diseño Orientado a Objetos.* Madrid : Universidad Politécnica de Madrid, 2005.

43. **PALACIO BAÑARES, Juan. 2007.** *Flexibilidad con Scrum. Principios de diseño e implementación de campos de Scrum.* 2007. Adaptando los procesos a la empresa.. 0710210187520.

44. **—. 2008.** *Navegápolis 2008.* Edición Febrero 2008. s.l. : Navegápolis®, 2008. 0802020409715.

45. **PALACIO, Juan. 2007.** *Flexibilidad con Scrum. Principios de diseño e implementación de campos de Scrum.* 2007. Adaptando los procesos a la empresa.. 0710210187520.

46. **PÉREZ MIRIÑÁN, Martín. 2002.** Asociación JavaHispano. *Asociación JavaHispano.* [En línea] Asociación JavaHispano, 17 de Diciembre de 2002. [Citado el: 9 de Febrero de 2008.] http://www.javahispano.org/contenidos/es/el_peligro_de_los_patrones/.

47. **PRESSMAN, ROGER S. 2001.** *Ingeniería del Software. Un enfoque práctico.* [trad.] Darrel Ince. Quinta Edición. Madrid : McGraw Hill/Interamericana de España, 2001. Adaptación al Español..

48. **PUIG, Vicenç, y otros. 2004.** *Control tolerante a fallos. Fundamento y Diagnóstico de fallos.* Departament Enginyeria de Sistemes, Automàtica e Informática Industrial (ESAII), Universidad Politècnica de Cataluña. España : s.n., 2004. pág. 34.

49. **R. CHIDAMBER, Shyam y F. KEMERER, Chris. 1994.** A Metrics Suite for Object Oriented Design. [aut. libro] IEEE. *IEEE Transactions on Software Engineering.* EE.UU. : IEEE Press Piscataway, NJ, USA, 1994, Vol. 20, págs. 476 - 493.

50. **Rational Software Corporation. 2003.** Ayuda extendida de RUP (Rational Unified Process). Versión 2003.06.00.65 *Rational Unified Process.* s.l. : Rational Software Corporation, 2003.

51. **REIMER, Darrell y SRINIVASAN, Harini. 2003.** *Analyzing Exception Usage in Large Java Applications.* IBM Research, IBM. New York : s.n., 2003. Artículo de Conferencia. Conferencia Internacional ECOOP'2003 (Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms).

52. **RENZEL, Klaus. 2003.** *Error Handling for Business Information Systems. A Pattern Language.* Primera Edición. s.l. : software design & management (sd&m), 2003. pág. 132.

53. **ROSANIGO, Zulema Beatriz. 2000.** *Maximizando reuso en software para Ingeniería Estructural. Modelos y Patrones.* Argentina : s.n., 2000. pág. 113. Tesis de Maestría en Ingeniería de Software.

54. **SCHMULLER, Joseph. 2000.** *Aprendiendo UML en 24 horas.* [ed.] Oscar MADRIGAL MUÑIZ. Primera Edición. México : Pearson Educación, 2000. pág. 448. 968-444-463-X.

55. **SHAW, Mary, y otros. 2007.** *First Steps toward a Unified Theory for Early Prediction of Software Value.* Proc. IEEE Equity, IEEE. Amsterdam : s.n., 2007. Artículo de Conferencia. Conferencia Internacional IEEE EQUITY 2007.

56. **SIEDERDLENBEN, Johannes. 2003.** *Errors and Exceptions – Rights and Responsibilities.* SD&M Research. Munich : s.n., 2003. Artículo de Conferencia. Conferencia Internacional ECOOP'2003 (Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms).

57. **VÉLEZ SERRANO, , José F., y otros. 2005.** *Técnicas avanzadas de diseño de software: Orientación a objetos, UML, patrones de diseño y Java.* Madrid : Universidad Rey Juan Carlos, 2005. pág. 171. Vol. 1.

58. **W. COOPER, James. 1998.** *The Design Patterns .* s.l. : Addison-Wesley, Design Patterns Series, 1998.

59. **WIPER, M.P. y WILSON, S. 2003.** *Comprobación de Softwrae. Un análisis Bayesiano.* Departamento de Estadística y Econometría, Universidad Carlos III de Madrid. España : s.n., 2003. Artículo de Investigación. 27 Congreso Nacional de Estadística e Investigación Operativa.

60. **ZARAZAGA, F.J., y otros. 2005.** *Automatizando el paso de Diseño Orientado a Objeto a Codificación mediante el uso de Metainformación basada en facets.* Departamento de Informática e Ingeniería de Sistemas, Centro Politécnico Superior, Universidad de Zaragoza. 2005. pág. 12.