CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS
AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

SECCIÓN DE COMPUTACIÓN

Análisis para el Proceso de Diseño de Sistemas

Tesis que presenta

José Jaime López Rabadán

Para obtener el grado de

Maestro en Ciencias

en la Especialidad de

Ingeniería Eléctrica

opción

Computación

Director de la tesis:

Dr. Pedro Mejía Alvarez

México, D.F. Agosto 2005

.

Resumen

La Ingeniería de Software estudia los diferentes procesos que se usan para la creación de software con calidad. Sus principales actividades son: análisis, diseño, implementación, validación, e incluso, el mantenimiento del software. En cada una de estas etapas se utilizan diferentes procesos que garantizan la calidad de un producto de software. Una de las etapas importantes es: el proceso de diseño, en el que se planea y se define cómo se debe implementar el sistema cuidando los diferentes recursos con los que se cuenta.

En este trabajo de tesis se describe lo que es la ingeniería de software para contextualizar el proceso de diseño. Además se describe el Proceso Unificado de Rational (RUP), el cual trabaja con un modelo iterativo, que disminuye los riesgos en la creación del software. El RUP, en la etapa de diseño ocupa el Lenguaje de Modelado Unificado (UML) para generar una serie de documentos que permiten la implementación de sistema.

Proponemos una guía para el diseño de sistemas, esta se basa en las etapas del diseño de sistemas y en el Proceso Unificado de Rational, con el objetivo de definir, de forma específica, las actividades que se desarrollan durante el proceso de diseño de un sistema. Además, se cuidan detalles de calidad en el diseño tales como: trazabilidad, mantenibilidad y evolutividad.

Esta guía se usó en un caso de estudio de un sistema denominado Sistema de Inscripciones Virtuales (SIV), que permite la inscripción de alumnos del CINVESTAV a través de la internet.

Abstract

The software engineer studies the different processes used for the creation of quality software. Their main activities are: analysis, design, implementation, validation even the maintaining of software. In each of these stages different processes are used that guarantee the quality of software products. One important stage is, the design process, where the system implementation is planed and defined, taking in account the different resources.

These thesis describes what the software engineer is for giving the design process context into the software engineer. Besides, the Rational Process Unified (RUP) is described, whitch works with the iterative model that minimize the risk in the software creation. The RUP, in its design stage uses the Unified Modeling Language (UML) for generating a serie of documents that allow the system implementation.

This work proposes a guide for the designing of systems, this guide is based on the system design and on the RUP, with the aim of defining in a specific form, the activities developed during the design process of a system. Besides, detailes are taken in account in the design such as: traceability, maintainability and evolutionary.

This guide was used in a case of study for a system denominated Sistema de Inscripciones Virtuales (SIV), which allows the CINVESTAV's student inscription through the Internet.

Dedicatoria

Este trabajo está dedicado a Dios, a mi familia y a cada uno de mis amigos...

Agradecimientos

- A Dios, quién me ha permitido llegar hasta este momento.
- A mi esposa, Araceli Hernández Hernández, por su apoyo, comprensión y por ser una mujer que a me ha ayudado a tener sueños y metas.
- A mi hija, Jeanete Xiadany López Hernández, por que es un encanto de niña y una motivación para mi vida y mi hijo que viene en camino.
- A mi Papá, José Jaime López Peña y Mamá Ma. del Carmen Rabadán Miranda, por que siempre han estado con migo en todo momento y me han apoyado.
- A mis Hermanos Fernando Isaac, Marco Antonio y Christian Natanael, por ser tres personas especiales en mi vida por su apoyo y porque siempre están cuando los necesito.
- A Jorge David Díaz Nava, Por enseñarme a no quedarme donde estoy y siempre luchar por lo que uno quiere, por su ejemplo y por ser alguien especial en mi vida.
- A mis amigos con los que he compartido momentos gratos.
- A mi Asesor el Dr. Pedro Mejía Álvarez, quien me dirijio en esta tesis y me compartio de sus conocimientos.
- A Sofi la secretaria de la Sección, por su apoyo.
- A mis revisores de tesis el Dr. Adriano de Luca Pennacchia y al Dr. Sergio Víctor Chapa Vergara.
- A los compañeros administrativos de la biblioteca del departamento de Ingeniería Eléctrica.
- Al Centro de Investigaciones y Estudios Avanzados del IPN, CINVESTAV-IPN junto con la sección de Computación del departamento de Ingeniería Eléctrica por ofrecerme las instalaciones y recursos necesarios para realizar este trabajo de investigación.

- Al Consejo Nacional de Ciencia y Tecnología, CONACyT, por el apoyo económico otorgado en el periódo agosto 2003- agosto 2004

.

Índice general

Capítulo 1

Introducción

El termino *Ingeniería de Software* surge en 1968 en la conferencia llamada *"NATO SOFTWARE ENGINEERING CONFERENCE 1968"* donde surge el termino *" crisis del software"*, En esta conferencia donde participaron 50 personas profesionales y catedráticos en el diseño de software de 11 diferentes países se concluía que los sistemas de software se construían ineficientemente, que excedían el presupuesto, que la mayoría de los sistemas que se planeaba realizar, eran de mala calidad o quedaban inconclusos, excedían por mucho el presupuesto programado, era casi imposible darles mantenimiento y mucho menos podían evolucionar junto con las necesidades diarias.

El uso de sistemas computacionales ha ido creciendo rápidamente al paso de los años. Hoy en día se puede observar que pequeñas, medianas y grandes empresas; áreas de educación, etc, necesitan la creación de nuevos sistemas computacionales, que cuenten con diseños adecuados, que permitan: el mantenimiento de los sistemas computacionales, estimar el costo y el tiempo para la construcción de un sistema así como su evolución.

La Ingeniería de software es una disciplina que proporciona a los desarrolladores y creadores de software, un conjunto de procedimientos y técnicas para llevar a cabo la especificación, análisis, diseño, implementación validación, e incluso el mantenimiento del software.

Con la aplicación de la Ingeniería de Software se pueden disminuir los riesgos de fallos

en un sistema de desarrollo e incrementar las posibilidades de la entrega del producto en el tiempo y costo estimado con calidad. Generalmente las etapas utilizadas en el desarrollo de software son: Ingeniería de requerimientos, Diseño del sistema, Implementación, Validación y Mantenimiento.

La etapa del diseño de sistemas, es una etapa importante, que definirá la arquitectura del sistema, la especificación abstracta del sistema, el diseño de las interfaces, de los componentes, de la estructura de datos y de los algoritmos. Por lo que ésta etapa, es básica para una implementación adecuada del sistema.

Para realizar el diseño de un sistema se pueden ocupar metodologías (*"como el Proceso Unificado de Rational"*) y lenguajes que expresen el diseño de los sistemas(*"como el Lenguaje de Modelado Unificado"*) que ayudan a organizar y estructurar el diseño de un sistema.

1.1. Motivación

La etapa del Diseño de sistemas, es determinante en un proyecto, dado que de ella depende la creación del sistema, los recursos que se invertirán en su creación y su futuro mantenimiento y evolución. La razón de elegir este tema se fundamenta en lo siguiente:

- La escasa importancia que se le da a la Ingeniería de Software tanto así que es frecuente observar, que se intenta desarrollar sistemas sin un diseño apropiado.
- Un gran número de proyectos son abandonados antes de ser culminados y cierto núemero de proyectos que son terminados, están fuera de presupuesto y tiempo.
- Muchos proyectos que son terminado, no pueden recibir mantenimiento, ni pueden evolucionar por que su diseño no lo permite ("cuando existe el diseño").
- Existen problemas entre la gente que desarrolla sistemas y la que los solicita, por que que los sistemas no cumplen con la funcionalidad esperada.

1.2. Planteamiento del problema

El diseño de sistemas es una disciplina donde se establece un conjunto de actividades, para traducir los requerimientos de un sistema a un documento, que especificará como un sistema se debe implementar.

El procesos de diseño es complejo y existen muchas metodologías para hacer diseños de sistemas, pero pocos ejemplos que muestren como utilizar una metodologías de forma práctica. Dado que el diseño de sistemas produce documentos, es necesario ocupar un cierto lenguaje accesible a los analistas, desarrolladores, documentadores y expertos de los modelos del negocios, que exprese de forma clara y no ambigua el diseño del sistema.

Por lo que en esta tesis se estudiara cada una de las etapas del diseño de sistemas y se revisara el Proceso Unificado de Rational (RUP) para proporcionar una guía para la realización del diseño de sistemas con calidad, expresando estos diseño con el Lenguaje de Modelado Unificadoe (UML).

1.3. Objetivos de la Tesis

Realizar un análisis del proceso de diseño, y proponer una guía para la creación de diseños con calidad. Ocupando como caso de estudio: El sistema de Inscripción Virtual a cursos de la sección de computación de Ingeniería del CINVESTAS (SIV).

1.3.1. Objetivos Particulares

- Definir las actividades que se desarrollan en el proceso de diseño.
- Demostrar el impacto que tiene un diseño en el desarrollo de un sistema.
- Identificar las ventajas y desventajas del uso de una metodología para el diseño de un sistema.

- Evaluar el impacto del uso de un lenguaje para expresar el diseño de un sistema (UML).

1.4. Organización de la Tesis

En el Capítulo 2 se definen conceptos de Ingeniería de Software que proporcionan un contexto general y permiten la ubicación del tema central de esta investigación. En el capítulo 3 se describe el proceso de diseño de sistemas, se definen las actividades y tareas que se deben realizar para diseñar un sistema. Se explican los diferentes documentos que se deben de generar y los criterios que se ocupan para diseñar un sistema. En el Capítulo 4 Se explica el Proceso Unificado de Rational partiendo de su Arquitectura la cual se divide en el modelo estático y dinamico. En el Capítulo 5 se describe el Lenguaje de Modelado Unificado y 7 de los principales diagramas de UML poniendo especial detalle en los diagramas de casos de uso, clases y secuenciales. En el Capitulo 6 se plantea el caso práctico haciendo uso del RUP y UML para la generación del diseño del Sistema de Inscripciones Virtuales (SIV).

y la aplicación de las actividades de diseño de sistemas así como el resultado obtenido, que es el documento de diseño del SIV

En el capítulo 7 se presentan las conclusiones de esta investigación y los trabjos futuros a desarrollar.

Capítulo 2

Conceptos básicos de Ingeniería de Software

La construcción de software requiere de la aplicación de un proceso de desarrollo que permita garantizar la calidad de los productos generados y que satisfaga las necesidades de las personas que los usaran. Generalmente, se aplica un enfoque de la Ingeniería de Software para producir software y el proceso de desarrollo a utilizar depende del contexto del sistema que se este construyendo. Los productos que se obtienen a través de este proceso son documentos, programas y archivos de datos que producen los Ingenieros de Software mediante las actividades de la ingeniería de software. Podemos enfatizar que el proceso, las técnicas, los métodos y las herramientas comprenden la Ingeniería de Software. En este capítulo se establecen algunos conceptos importantes de Ingeniería de Software que nos permitan comprender y ubicar las actividades de la Ingeniería de Requerimientos en el proceso de desarrollo del software.

2.1. ¿Qué es la Ingeniería de Software?

Actualmente los sistemas computacionales están presentes en todas las áreas del conocimiento, por lo que estos sistemas deben garantizar la confiabilidad de los datos almacenados y

la información producida. Los errores en el software pueden causar defectos en los datos y en la información generada, así como en las decisiones que tomen los usuarios que operan el sistema. En consecuencia, los defectos de software pueden producir pérdidas económicas en el negocio, daños en el medio ambiente o incluso en sistemas críticos, las fallas pueden provocar la muerte de las personas que hacen uso de estos sistemas. Es preciso señalar que el desarrollo de software debe ser un proceso controlado y planeado, en el que deben tomarse en cuenta las necesidades y sugerencias de los usuarios o clientes quienes finalmente son los que más conocen el funcionamiento de su negocio. La utilización de modelos de representación en el proceso de desarrollo de software permite reducir la probabilidad de riesgos y fallas, además garantizan la entrega en tiempo y forma del sistema al cliente. Los productos son entregados con calidad y dentro de los costos estimados. Se puede definir a la Ingeniería de Software como una disciplina que proporciona un conjunto de procedimientos y técnicas para la producción y desarrollo sistematizado, disciplinado y cuantificable de aplicaciones de software, que satisface los requerimientos de funcionalidad y desempeño[]. Generalmente como expertos en el área de la computación nos dedicamos a analizar problemas relacionados con un sistema de computación existente, por lo tanto, es necesario estudiar el ambiente del sistema, para no imponer técnicas de computación en todo problema que surja. Primero debe resolverse el problema, después si es necesario, se puede utilizar la tecnología computacional como herramienta para implementar nuestra solución.

2.2. ¿Qué hacen los Ingenieros de Software?

Las computadoras y los lenguajes de programación, pueden verse desde dos puntos de vista: como objetos de estudio y como herramientas a utilizar en el diseño e implementación de una solución a un problema. Los Ingenieros de Software, en lugar de investigar el diseño del hardware ó de probar teorías acerca de cómo trabajan los algoritmos, trabajan con las funciones de un sistema de cómputo como parte de una solución general. Utilizan todos sus conocimientos técnicos de computación para tratar de resolver problemas previamente estudiados y que

pueden ser factiblemente computarizados mediante técnicas y herramientas de la Ingeniería de Software. El desarrollo de un sistema involucra a personas que son integradas en equipos y que desarrollan tareas específicas. Se conoce con el nombre de desarrolladores a los ingenieros de software debido a que desempeñan diferentes papeles en el proceso de desarrollo de software. En el equipo de desarrolladores de software se pueden encontrar ingenieros especializados en cada una de las actividades, por lo tanto deben existir analistas de requerimientos, quienes trabajan con los clientes desglosando en requerimientos las necesidades de los clientes. Una vez conocidos y documentados los requerimientos, los analistas de requerimientos trabajan con los diseñadores para generar una descripción a nivel de sistema de lo que el sistema debe hacer. Los diseñadores trabajaran con los programadores para describirles el sistema en una forma tal que dichos programadores implementen lo que los requerimientos especifican. Posteriormente, el código debe ser probado, y en ocasiones las primeras pruebas la hacen los mismos programadores, aunque a veces intervienen probadores adicionales para ayudar a detectar defectos que los programadores pasan por alto. Después de comprobarse la funcionalidad y calidad del sistema se trabaja con el cliente para verificar que el sistema completo resulta en lo que el cliente desea. Esto se hace comparando cómo trabaja el sistema en relación al conjunto inicial de requerimientos. Finalmente, los entrenadores enseñan al usuario como se utiliza el sistema. Para muchos sistemas de software, la aceptación por parte del cliente no significa la finalización de las tareas de desarrollo, si existen defectos después de la aceptación del sistema, intervendrá un equipo de mantenimiento para corregirlos. De cualquier forma, los requerimientos del cliente pueden cambiar a medida que transcurre el tiempo y deberán hacerse los cambios correspondientes al sistema. El mantenimiento puede implicar a analistas para determinar qué requerimientos se agregan o cambian, a diseñadores para establecer en qué lugares del diseño del sistema deben hacerse los cambios, a programadores para implementar los cambios, a probadores para asegurar que el sistema cambiado funciona correctamente y a entrenadores para explicar a los usuarios la forma en que los cambios afectan el uso del sistema [].

2.3. El Software

Cuando hablamos de software, muchas veces nos referimos a programas de computadoras, pero el software no sólo son programas, sino todos los documentos generados, la configuración ó la estructura de los datos que se necesitan para hacer que estos programas funcionen de manera correcta. En general, un sistema de software consiste de varios programas independientes, archivos de configuración, documentación que describe la estructura del sistema, planes y manuales de usuario que explica como usar el sistema. En la literatura de Ingeniería de Software [],se distinguen dos tipos de productos de software.

1. Software genérico. Son sistemas aislados producidos por una organización de desarrollo que se vende al mercado abierto a cualquier cliente que le sea posible adquirirlo. La organización de desarrollo controla la especificación del producto.

2. Software personalizado. Son sistemas requeridos por un cliente en particular. Una empresa de software desarrolla un sistema especial para un cliente. Por lo general, la especificación de este producto es controlada por el cliente del software, y los desarrolladores deben trabajar con esa especificación.

Las aplicaciones de software ya sean genéricas o especializadas, se encuentran dentro de alguna de las siguientes clasificaciones [24].

1. Independientes. Es software que reside en una sola computadora, además de que no se conecta con otro software o hardware.

2. Inmersos. El software es parte de una aplicación única que incluye hardware.

3. De tiempo real. La aplicación de software debe ejecutar funciones en un límite de un tiempo corto, generalmente en milisegundos.

4. De redes. El software consiste en componentes que interactúan utilizando los beneficios de una red.

2.4. Atributos de un buen software

Un software de alta calidad es aquel que satisface las necesidades del usuario y posee las siguientes características:

- Útil y aprovechable. Es decir, hace el trabajo de los usuarios más fácil o mejor.
- Fiable. Un software debe contener pocos errores o idealmente ninguno.
- Flexible. En el transcurso del desarrollo del software, las necesidades y requerimientos de los usuarios cambian, estos cambios deben poder realizarse en el momento que se requieran. A los cambios que son realizados después de la entrega del sistema, se les conoce como mantenimiento.
- Accesible. La venta como el costo del mantenimiento debe estar en función de la mano de obra y el alcance del sistema.
- Disponible. La disponibilidad del software consiste en dos aspectos: primero, tiene que poder ejecutarse con el equipo de cómputo disponible y debe ser suficientemente portable. Segundo, un proyecto de software debe culminar con la entrega del sistema prometido.

2.5. Defecto, error y falla en el software

A pesar del gran avance que hay en las técnicas para la Ingeniería de Software siguen existiendo sistemas que presentan problemas. Los defectos en el software en algunos casos solo causan molestias, pero en otros cuestan grandes cantidades de tiempo y dinero, y en ocasiones son una amenaza para la vida. En la 2.1 se muestra la relación que existe entre defectos, errores y fallas.

Se produce un defecto cuando una persona tiene una equivocación denominada error, al realizar alguna actividad concerniente al software. Por ejemplo, un diseñador puede comprender mal un requerimiento y crear un diseño que no corresponda con la interpretación del analista y de las necesidades del usuario, de esta forma, el defecto en el diseño puede propagarse en el momento de que los programadores traduzcan el diseño a código. Esto puede dar lugar a otros

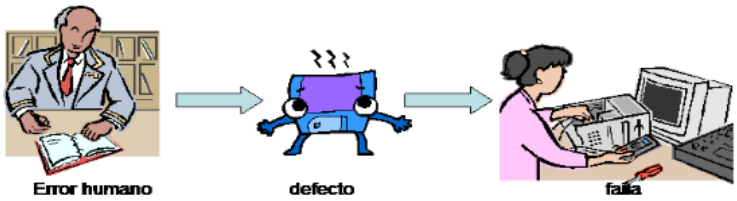


Figura 2.1: Relación entre defecto, error y falla.

defectos, tales como código incorrecto y una descripción incorrecta en el manual del usuario. De esta manera un error puede generar múltiples defectos y un defecto puede encontrarse en cualquier producto en desarrollo o mantenimiento. Una falla es un desvío respecto al comportamiento requerido del sistema, puede descubrirse antes o después de la entrega del sistema, durante la prueba, la operación o durante el mantenimiento. Una falla indica que el sistema no esta funcionando como se ha especificado [24]. Un defecto es una vista interna del sistema desde la perspectiva de los desarrolladores, mientras que una falla es una vista externa o un problema que ve un usuario, sin embargo, no todos los defectos llevan a producir una falla. Por lo tanto, deben buscarse mecanismos que minimicen el número de errores en la etapa de especificación de los requerimientos para evitar futuros defectos y fallas. Esto se puede lograr empleando técnicas, métodos y herramientas adecuadas al dominio del sistema en el proceso de desarrollo del software.

2.6. Elementos en el desarrollo de software

La elaboración de un proyecto de software, implica la intervención de diferentes tipos de personas guiadas mediante un proceso de desarrollo para obtener un producto de software. Generalmente, en el proceso de desarrollo se utiliza un conjunto de herramientas para automatizar el proceso y facilitar las tareas al equipo de desarrollo. Podemos decir que estos son los elementos principales que intervienen en la manufactura de software. [24]

1. Personas. Son el elemento principal y son los autores de un proyecto de software y generalmente comprende a los arquitectos, analistas, desarrolladores, Ingenieros de prueba y al personal de soporte, además de los clientes, usuarios y otros interesados.

2. Proyecto. Es el elemento organizativo mediante el cuál se gestiona el desarrollo de software. El resultado es la versión final de un producto.

3. Producto. Comprende a todos los artefactos que se crean durante el desarrollo del proyecto (documentos, modelos, diagramas, código fuente, código ejecutable y pruebas).

4. Proceso. Son un conjunto de actividades necesarias para transformar las necesidades del usuario en un producto.

5. Herramientas. Es el software que se utiliza para automatizar las actividades del proceso.

2.7. El proceso de desarrollo de Ingeniería de Software

El proceso de desarrollo de software es una descripción de la construcción del software que contiene actividades organizadas de modo que en conjunto producen código probado. No existe una definición estándar de estas actividades y muchos autores le dan importancia a algunas más que a otras. Generalmente, podemos clasificar estas actividades en:

1. Ingeniería de requerimientos.

2. Diseño del sistema.

3. Implementación del software.

4. Prueba o validación del software.

5. Mantenimiento.

En la Ingeniería de requerimientos, es necesario conocer la naturaleza del sistema, entender lo que desean los clientes, delimitar el alcance del sistema teniendo en cuenta el tiempo disponible, el presupuesto y el personal asignado. El producto generado de esta etapa, es un documento de especificación de requerimientos en donde se encuentran definidos todos los servicios requeridos del sistema y las restricciones sobre las que debe operar. El diseño del sistema toma como base el documento de especificación de requerimientos, agrega detalles a cada requerimiento, identifica los subsistemas, describe la estructura interna de los datos y las interfaces entre los componentes del sistema. El diseño del sistema utiliza varios modelos para la representación del sistema desde diferentes perspectivas y niveles de abstracción. El resultado de esta etapa es la especificación precisa de los algoritmos y estructuras de datos que van a implementarse. En la etapa de implementación se lleva a cabo la codificación del sistema, se deben satisfacer los requerimientos de la manera que especifica el diseño detallado. El programador examina los documentos generados en las etapas anteriores para evitar inconsistencias entre los documentos. Se toman en cuenta los estándares de programación, así como los lenguajes de programación. Las pruebas y la validación se utilizan para demostrar que el sistema cumple con su especificación y satisface las necesidades del usuario. De cualquier forma, es necesario llevar a cabo un proceso de verificación por medio de inspecciones y revisiones desde la especificación de los requerimientos hasta la puesta en marcha del sistema. El propósito de realizar pruebas no es demostrar que una aplicación es satisfactoria, sino determinar con firmeza en qué parte no lo es. Las pruebas nos permiten mostrar la presencia de defectos en el sistema. El mantenimiento de software consiste en las actividades realizadas sobre el sistema una vez entregado y puesto en marcha. Una definición es [24]. “El proceso de modificar un sistema o componente de software entregado para corregir defectos, mejorar el desempeño, mejorar algún atributo, o adaptarlo al cambio de entorno”. Como se observa, el resultado de la Ingeniería de Software consiste en mucho más que el código del sistema, es decir, incluye planes, informes, documentos e inclusive programas llamados prototipos que son desechados después de lograr el objetivo por lo que fueron creados.

2.8. Paradigmas del proceso de desarrollo de software

Un paradigma o modelo del proceso de desarrollo de software, es una descripción de las actividades que se llevan a cabo en la elaboración de un producto de software. Al proceso completo de desarrollo, también se le conoce como ciclo de vida del software, debido a que en él se detalla la vida de un sistema desde su concepción, implementación, entrega, utilización y hasta su mantenimiento. En la literatura de Ingeniería de Software existe una variedad de modelos o paradigmas de desarrollo. Los modelos más utilizados se describen a continuación.

2.8.1. Modelo de desarrollo en cascada

El "modelo en cascada", define un enfoque secuencial del proceso de desarrollo, separando en etapas y cada etapa es fundamental en el desarrollo (figura 2.2). Cada etapa debe completarse antes de comenzar la siguiente etapa. El problema que presenta este modelo es debido a la separación de las actividades en etapas, en cada fase el resultado es uno o más documentos aprobados. En la práctica estas actividades del desarrollo de un producto de software están entrelazadas y requieren de continuas iteraciones, además si un error no es detectado al principio de la etapa, puede ser desastroso encontrarlo en etapas posteriores. Es difícil dejar definidos los requerimientos en la primera etapa del desarrollo, como lo expresa el modelo, esto podría generar problemas a la hora de encontrar nuevos requerimientos en las últimas etapas y no fuera posible regresar a la primera etapa para agregarlos. Para el modelo en cascada el cliente podría tener una versión operativa del sistema sólo hasta alcanzar las etapas finales del desarrollo [24].

2.8.2. Desarrollo Evolutivo

Este paradigma recomienda el desarrollo de una implementación inicial del sistema, que debe ser presentada al cliente para su negociación, y posteriormente refinada mediante versiones hasta alcanzar el desarrollo completo del sistema. Este modelo de desarrollo es más conocido como prototipado, en donde las actividades son llevadas a cabo en forma concurrente y tienen

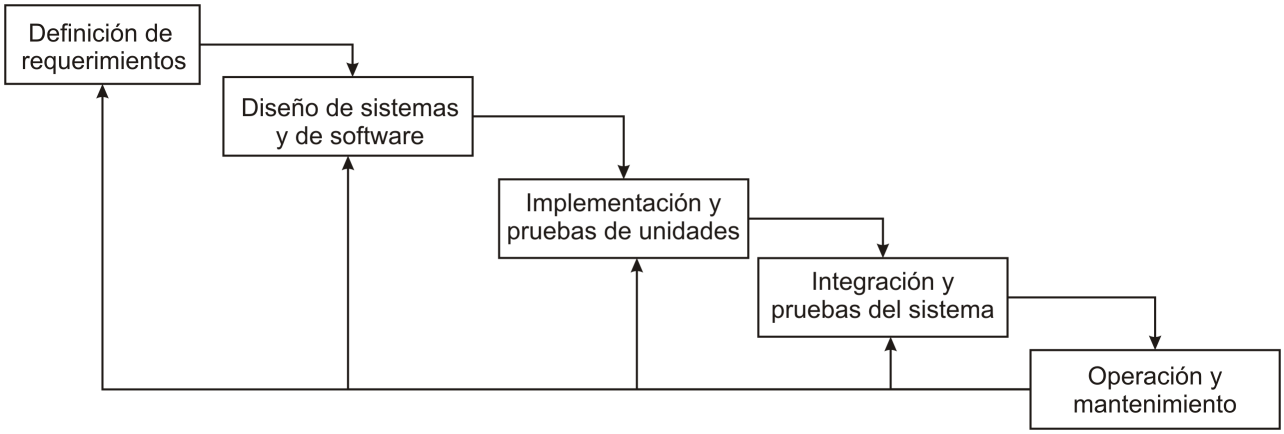


Figura 2.2: Modelo en cascada.

retroalimentación en todo el proceso.

Existen dos tipos de desarrollos evolutivos:

1. Desarrollo Exploratorio. En este tipo de desarrollo se trabaja con el cliente para encontrar sus requerimientos y entregar un sistema final. El sistema es mejorado cada vez que existan nuevas propuestas del cliente.

2. Prototipos Desechables. El objetivo de los prototipos desechables es comprender los requerimientos del cliente. Una vez logrado el objetivo se desecha el prototipo actual para construir otro que permita comprender mejor los requerimientos.

La principal ventaja de este enfoque se basa en que la especificación puede desarrollarse en forma creciente, conforme los usuarios adquieran un mayor conocimiento de su problema. Sin embargo, este enfoque también presenta algunas deficiencias:

- El proceso no es visible. Se tienen que realizar entregas regulares de versiones del sistema para medir el progreso, y es muy costoso producir documentos para cada versión del sistema.
- Generalmente se tiene una estructura deficiente. Los cambios continuos en el software tienden a descomponer la estructura del sistema, además su aplicación es una tarea difícil y costosa.

- Se requieren herramientas y técnicas especiales que permitan un desarrollo rápido y que no presenten incompatibilidades entre sí.

2.8.3. Modelo de desarrollo Incremental e Iterativo

Una manera de reducir el tiempo de entrega de un sistema, es mediante la utilización del desarrollo por fases. El sistema es diseñado de tal forma que pueda ser entregado por módulos, lo que permite que los usuarios dispongan de su parcial funcionalidad a medida que el sistema se encuentra en desarrollo [24]. Los dos enfoques más conocidos de este modelo son:

1. Desarrollo incremental.

2. Desarrollo iterativo.

En el Desarrollo incremental el sistema es fragmentado en subsistemas de acuerdo a su funcionalidad y de como este definido en el documento de requerimientos. Las versiones son definidas iniciando con un módulo funcional pequeño y con cada nueva versión se agrega funcionalidad al sistema total (figura 2.3). Uno de los problemas que presenta este enfoque es la dificultad en la identificación de los recursos comunes que requieren todos los incrementos. Esto se debe a que los requerimientos no se definen en detalles hasta que un incremento se implemente.

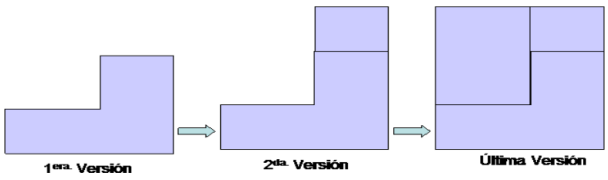


Figura 2.3: Modelo incremental.

En el Desarrollo iterativo el sistema es entregado completo, aunque la funcionalidad sea primitiva, es decir, que las partes del sistema no este funcionando totalmente. Posteriormente

cada parte del sistema es reforzado con calidad e implementándose las funcionalidades faltantes en las versiones anteriores (figura 2.4).

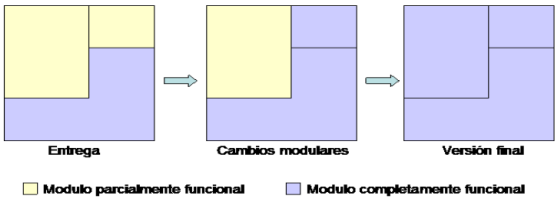


Figura 2.4: Modelo iterativo.

2.8.4. Desarrollo en espiral

El modelo en espiral fue propuesto por Boehm en el año de 1988. Este modelo representa el proceso de software como una espiral, examina el progreso de desarrollo de software tomando en consideración los riesgos involucrados para minimizarlos y controlarlos, de esta forma combina las actividades del desarrollo con la gestión del riesgo [24]. El espiral se divide en cuatro sectores que comprenden: la definición de objetivos, la evaluación y reducción de riesgos, el desarrollo y validación, y por último la planeación (figura 2.5).

El modelo del espiral se ajusta al avance de los proyectos, sin embargo requiere de una administración mucho más cuidadosa que la del modelo en cascada. El ciclo del espiral comienza con los requerimientos y un plan de desarrollo, agregando una evaluación de riesgos y construye prototipos alternativos antes de escribir el documento de conceptos de operación. El documento de conceptos de operación describe en un alto nivel como debe trabajar el sistema, es decir, especifica un conjunto de requerimientos completos y consistentes. El principal producto de la primera iteración es el concepto de operaciones, en la segunda iteración son los requerimientos, en la tercera iteración el diseño y de la cuarta son las pruebas. Para cada iteración el análisis de riesgos pondera diferentes alternativas en base de los requerimientos y restricciones, el prototipado verifica el grado de factibilidad antes de elegir alguna alternativa de desarrollo en

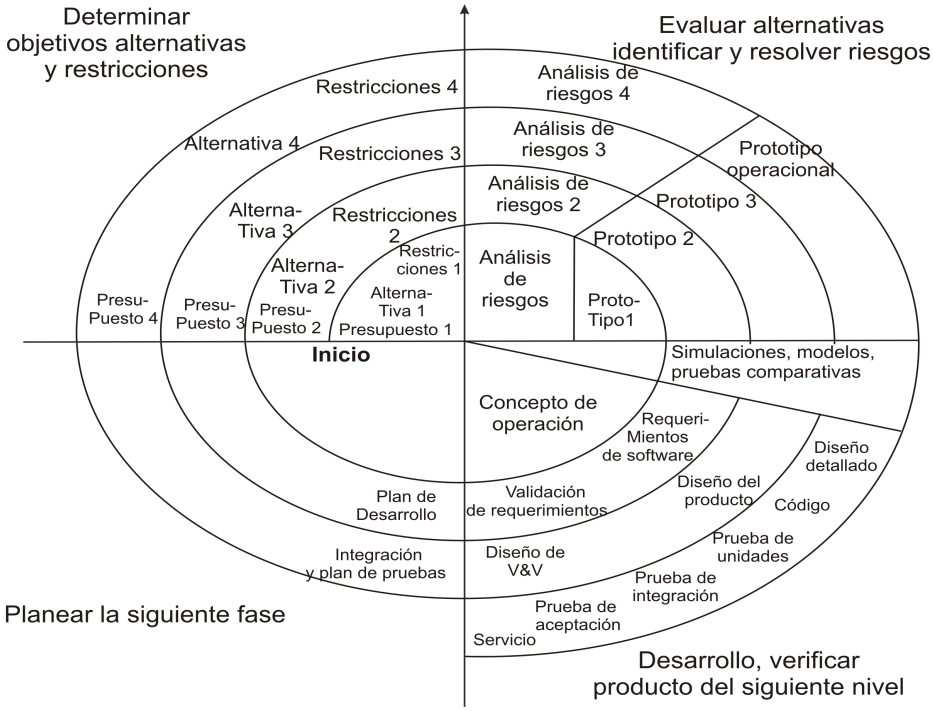


Figura 2.5: Desarrollo en espiral.

particular, si se identifican riesgos los lideres del equipo de desarrollo son los que deciden como eliminarlos o minimizarlos [24].

2.8.5. Modelos de Métodos Formales

Comprenden un conjunto de actividades que conducen a la especificación matemática del software. Los métodos formales permiten a los ingenieros de software especificar, desarrollar y verificar un sistema, aplicando una notación matemática. El enfoque más conocido del proceso de desarrollo formal es “la ingeniería de la sala limpia”(clear room), desarrollado por IBM. A pesar de que estos métodos proporcionan mecanismos para reducir muchos de los problemas que son difíciles de superar con otros enfoques de la Ingeniería de Software, existen situaciones en torno a su aplicabilidad [24].

- El desarrollo de modelos formales actualmente es caro y consume mucho tiempo.

- La mayoría de los desarrolladores no tienen antecedentes necesarios para aplicarlos.
- No son recomendables utilizarlos como mecanismos de comunicación con los clientes del sistema, ya que no tienen conocimientos técnicos.

2.8.6. Desarrollo basado en componentes

Este enfoque se basa en la reutilización de componentes de software reutilizable. Generalmente esto sucede cuando los desarrolladores conocen diseños o código similares requeridos y que ya fueron elaborados para otros sistema [24]. Para garantizar el proceso de desarrollo, y además de las actividades de especificación de requerimientos y la validación del sistema, se integran nuevas actividades que son:

1. Análisis de componentes. En esta fase se buscan los componentes que satisfagan las especificaciones de los requerimientos del sistema a desarrollar.

2. Modificación de componentes. Se analizan tanto los requerimientos como los componentes, adaptando los componentes que lo requieran, y si no son posible estos cambios, se realiza nuevamente la búsqueda de nuevos componentes para buscar soluciones alternativas.

3. Diseño de sistemas con reutilización. En esta actividad, se diseña el sistema con los componentes encontrados y si no hay componentes disponibles se diseñan nuevos.

4. Desarrollo e integración. La integración de sistemas es parte del desarrollo y los componentes que no se puedan adquirir se desarrollan.

Con la reutilización podemos obtener ventajas en la reducción de la cantidad de software a desarrollar, de costos y riesgos, conduciendo a la entrega rápida del software. Sin embargo, al igual que todos los modelos, este también presenta inconvenientes. El principal inconveniente consiste en que la reutilización de componentes nos conduce a que un sistema no cumpla con

las necesidades reales de los usuarios definidas en el documento de requerimientos, esto se debe a los ajustes que se les hace a los componentes para adaptarlos al sistema en desarrollo.

2.9. Problemas que enfrenta la Ingeniería de Software

A pesar de los grandes avances que existen en la Ingeniería de Software, aún hay problemas que se presentan en el desarrollo de proyectos de software y que impiden su conclusión o retrasan su entrega. Algunos de los problemas que enfrenta la Ingeniería de Software son:

1. La necesidad de las empresas clientes, por presentar sus productos en el mercado en el menor tiempo posible.

2. La complejidad de los sistemas.

3. La dificultad de integrar cambios a sistemas con código obsoleto.

4. La difícil integración de un nuevo software a otros sistemas existentes.

5. La elaboración de software sin utilizar los procesos de desarrollo que garanticen la conclusión de los productos con calidad y sin alterar los costos con que fueron concebidos.

6. La introducción de nuevas técnicas y herramientas en el proceso de desarrollo de un sistema y que requieran de amplios conocimientos técnicos para su manejo.

7. El no descubrimiento de todos los requerimientos de software en etapas iniciales del desarrollo del sistema.

8. La naturaleza de cambio en que se encuentran las empresas clientes y que requiere que los sistemas se adapte a estos cambios.

9. La mala interpretación de los requerimientos por los desarrolladores.

10. La falta de entendimiento entre el equipo de desarrollo y los usuarios o clientes para homogenizar la comprensión de los requerimientos.

Estos son algunos de los problemas que podemos encontrar y que pueden ser los responsables del fracaso de muchos proyectos de software.

Estos son algunos de los casos más mencionados sobre fallos de sistemas, y que son generados por errores en algunas de las etapas del proceso de desarrollo de software.

Capítulo 3

Proceso de Diseño

El diseño puede definirse como el proceso que se sigue para bosquejar, describir y abstraer algo que ya existe o que se desea construir, a través de gráficos o texto con el objetivo de generar una guía que sirva para su creación.

El diseño de software abstrae los requerimientos de los usuarios para que, a través de un proceso de diseño se bosqueje y describa la estructura de lo que será un sistema a implementar. El diseño puede realizarse basándose en un modelo estructurado u orientado a objetos, y en la arquitectura con la que se desee trabajar. Para diseñar un sistema se debe seguir una serie de pasos estructurados que permitan llegar al objetivo de implementar cierto software. Existen un gran número de métodos para diseñar un sistema, sin embargo se ha observado que el proceso para diseño de software incluye las siguientes etapas [24]:

- Diseño Arquitectónico
- Especificación Abstracta
- Diseño de Interfaz
- Diseño de Componentes
- Diseño de la Estructura de Datos
- Diseño de algoritmos

El diseño de un sistema tiene ciertas características que determinan su calidad. Tales carac-

terísticas tienen que ver con la Usabilidad, Mantenibilidad, Portabilidad, Confiabilidad, Disponibilidad, Fiabilidad, Seguridad y Protección. Para determinar si el diseño es correcto, se deben considerar los siguientes aspectos:

- El diseño debe reflejar todos los requerimientos obtenidos durante el análisis del modelo de negocios.
- El diseño debe ser una guía legible y comprensible para aquellos que generan código, que realizan pruebas y dan mantenimiento al sistema.
- El diseño debe proporcionar una guía, para la construcción del software.

El Diseño de Software sirve para definir y detallar como se construirá el software, Estima el tiempo y el costo que tendrá la implementación del sistema, también, cuida el proporcionar la información necesaria para que el sistema evolucione y sea mantenible.

A continuación explicaremos cada una de las etapas del proceso de diseño así como las características de un diseño con calidad.

3.1. Diseño Arquitectónico

Al proceso de diseño que identifica los subsistemas y establece un marco de trabajo, para control y comunicación de lo subsistemas se le llama: *"diseño arquitectónico"*.

El diseño arquitectónico tiene como principal propósito bosquejar lo que será el sistema, pero con un alto nivel de abstracción, el proceso de diseño arquitectónico comprende el establecimiento de un marco de trabajo estructural básico para un sistema. Esto implica identificar los componentes principales del sistema y la comunicación entre ellos. Bass (1988) plantea tres ventajas de la especificación del diseño y la documentación de una arquitectura de software:

- *Comunicación entre los expertos del negocio*. La arquitectura es una presentación de alto nivel del sistema, que es utilizada como punto de discusión.

- *Análisis del sistema*. Hace explícita la arquitectura del sistema en una etapa inicial del desarrollo del sistema, lo cual significa que se debe llevar a cabo cierto tipo de análisis. Las decisiones en el diseño arquitectónico tienen alto impacto sobre todo cuando se desea cumplir con los requerimientos de desempeño, fiabilidad y de mantenibilidad.

- *Reutilización a gran escala*. Una arquitectura del sistema es una descripción compacta y manejable de cómo se organiza el sistema y como ínteroperan los componentes.

Existen diferentes formas de ver el diseño arquitectónico, sin embargo sin importar el enfoque existen tres actividades básicas:

- ***Estructuración del sistema***. Se identifican los subsistemas principales para proporcionar un esquema de la estructura del sistema por lo que también es necesario identificar la comunicación entre los subsistemas.

- ***Modelado de control***. Se establece un modelo general de las relaciones de control entre las partes del sistema.

- ***Descomposición modular***. Cada subsistema identificado se descompone en módulos y sus interconexiones.

Debemos aclarar que un subsistema y un módulo se distinguen en que un subsistema es, un sistema por sí mismo cuya operación no depende de los servicios suministrados por otro subsistema, por lo que el subsistema se compone de módulos los cuales tienen interfaces definidas para su comunicación con otros subsistemas, mientras que el modulo es un componente del sistema que suministra uno o más servicios.

El resultado del proceso arquitectónico es un documento de diseño arquitectónico que consiste de varias representaciones gráficas de los modelos del sistema junto con el texto descriptivo asociado.Así mismo describe como se estructura el sistema en subsistemas y cómo cada subsistema se estructura en módulos.

Los diversos modelos gráficos del sistema muestran diferentes perspectivas de la arquitectura, los modelos que se pueden desarrollar son:

1. *El modelo estructural estático* el cual muestra los subsistemas o componentes a desarrollar como unidades independientes.

2. *El modelo de procesos dinámicos* el cual muestra cómo se organiza el sistema en tiempo de ejecución.

3. *El modelo de interfaz* el cual define los servicios ofrecidos por cada subsistema a través de su interfaz pública.

4. *Los Modelos de relación* los cuales muestran las relaciones de, por ejemplo el flujo de datos entre los subsistemas.

La arquitectura del sistema afecta al desempeño, la robustez, la distributividad y la mantenibilidad de un sistema, por lo tanto, el estilo particular y la estructura elegida para una aplicación puede depender de los requerimientos no funcionales del sistema que los cuales describiremos a continuación:

1. *Desempeño* Si es un requerimiento crítico del sistema, esto sugiere, que la arquitectura se debe diseñar para localizar las operaciones críticas dentro de un número reducido de subsistemas con poca comunicación, hasta donde sea posible, entre estos subsistemas.

2. *Seguridad*: Si este es un requerimiento crítico del sistema, esto sugiere, utilizar una estructura en capas para la arquitectura con los recursos más críticos protegido por las capas más internas y con un alto nivel de validación de la seguridad aplicado a esas capas.

3. *Protección*: Si este es un requerimiento crítico del sistema, esto sugiere, que la arquitectura se debe diseñar de tal forma, que las operaciones relacionadas con la protección, se localicen en un solo subsistema o en un número reducido de subsistemas; esto reduce los

costos y los problemas de validación y hace posible crear sistemas de protección relacionados.

4. *Disponibilidad*: Si este es un requerimiento crítico del sistema, esto sugiere, que la arquitectura debe diseñarse para incluir componentes redundantes de tal forma que se posible, reemplazar y actualizar los componentes sin detener el sistema.

5. *Mantenibilidad*: Si este es Si es un requerimiento crítico del sistema, esto sugiere, que la arquitectura del sistema se debe diseña utilizando componentes auto contenidos de grano fino que pueden combinarse con facilidad. Los productores de datos deben estar separados de los consumidores y la estructura de datos compartidas deben evitarse.

3.1.1. Estructuración del Sistema

La primera fase de la actividad de diseño arquitectónico se refiere por lo general, a la descomposición del sistema en un conjunto de subsistemas que interactúan en el nivel más abstracto. Un diseño arquitectónico se puede ver como un diagrama de bloques en el que cada cuadro del diagrama representa un subsistema los cuadros dentro de otros cuadros indican que, el subsistema se ha descompuesto en subsistemas. Las flechas indican que los datos y/o el control se pasan de un subsistema a otro en la dirección de las flechas, por lo que un diagrama arquitectónico de bloques presenta un panorama general de la estructura del sistema.

Este tipo de diagramas tienen limitaciones dado que no muestran la naturaleza de las relaciones entre los componentes del sistema ni sus propiedades visibles externas. Sin embargo este tipo de modelos es efectivo para la comunicación con los especialistas del modelo de negocios y con los usuarios del sistema.

Los diagramas de cuadros no son la única representación arquitectónica que se utiliza. Existen modelos con propósitos más específicos par la descripción de la estructura del sistema como son:

- *El modelo de depósito*: Los subsistemas que componen un sistema deben intercambiar información con el fin de que puedan trabajar de forma conjunta y efectiva.

  Existen dos forma fundamentales para lograr esto:

  1. Todos los datos compartidos se ubican en una base de datos central que puede ser accedida por todos los subsistemas.
  2. Cada subsistema tiene sus propia base de datos, donde los datos se intercambian con otros subsistemas pasando mensajes entre ellos.

  La mayoría de los sistemas que utilizan grandes cantidades de datos se organizan alrededor de una base de datos compartida o depósito, este tipo de modelos es adecuado para aplicaciones donde los datos sean generados por un subsistema y utilizados por otro.

- *El modelo Cliente - Servidor*: Es un modelo de sistemas distribuido que muestra como los datos y el procesamiento se distribuyen a lo largo de varios procesadores, los componentes principales son:

  1. Un conjunto de servidores independientes que ofrecen servicios a otros subsistemas.
  2. Un conjunto de clientes que llaman a los servicios ofrecidos por los servidores, por lo que existen varias instancias de un programa cliente que se ejecuta de forma concurrente.
  3. Una red que permite a los clientes acceder a estos servicios.

  Los clientes tienen que conocer los nombres de los servidores disponibles y los servicios que suministran, sin embargo, los servidores no requieren conocer la identidad de los clientes o cuantos clientes existen. Los clientes acceden a los servicios suministrados por un servidor a través de llamadas a procedimientos remotos.

- *El modelo de máquina abstracta*: El modelo de maquina abstracta de una arquitectura, modela la interacción entre los subsistemas y organiza un sistema en una serie de capas

cada una de las cuales suministra un conjunto de servicios. Cada capa define una máquina abstracta cuyo lenguaje de máquina, se utiliza para implementar el siguiente nivel de máquina abstracta.

El enfoque de capas permite el desarrollo incremental de sistemas. Cuando una capa se desarrolla, algunos de los servicios suministrados por esa capa están disponibles para los usuarios. Esta arquitectura es cambiable y portable.

3.1.2. Modelos de control

Los modelos para estructurar un sistema describen la manera en que un sistema se descompone en subsistemas. Para trabajar como un sistema, los subsistemas deben controlarse para que sus servicios se entreguen al lugar correcto y en el momento justo. Los modelos estructurales no incluyen información de control, en su lugar el arquitecto debe organizar los subsistemas acorde a un modelo de control que complemente el modelo estructural que se utiliza.

Los modelos de control en el nivel arquitectónico comprenden el nivel de control entre los subsistemas. Se pueden identificar dos enfoques generales para el control:

1. *Control centralizado*: Es donde un subsistema se designa como controlador del sistema y tiene la responsabilidad de administrar la ejecución de otros subsistemas. Los modelos de control centralizado se dividen en dos clases, dependiendo de si los subsistemas controlados se ejecutan secuencialmente ó en paralelo.

2. *Sistemas dirigidos por eventos*: En los modelos de control centralizado, las decisiones de control por lo regular se determinan mediante los valores de algunas variables de estado del sistema, en contraste, los modelos de control dirigidos por eventos se rigen por eventos generados del exterior, donde estos eventos pueden provenir de otros subsistemas o del entorno del sistema.

   El término evento en este contexto no sólo significa una señal binaria sino que además, puede ser una señal que toma varios valores. Existen diferentes tipos de sistemas dirigidos

por eventos pero solo enunciaremos dos:

a) *Modelos de transmisión*: Un evento se transmite en principio a todos los subsistemas, cualquier subsistema que pueda manejar ese evento responde a él.

b) *Modelos dirigidos por interrupciones*: Se utilizan exclusivamente en sistemas de tiempo real donde las interrupciones externas son detectadas por un controlador de interrupciones, posteriormente son transferidas a otro componente para su procesamiento.

3.1.3. Descomposición Modular

Después de diseñar una arquitectura estructural, la siguiente etapa del proceso de diseño arquitectónico consiste en descomponer los subsistemas en módulos. No existe una distinción rígida entre la descomposición del sistema y la descomposición modular, sin embargo, por lo general los componentes dentro de los módulos son más pequeños que los subsistemas y esto permite utilizar modelos alternativos de descomposición.

Consideremos dos modelos que son de utilidad cuando se descompone un subsistema en módulos:

1. *Un modelo orientado a objetos*: El sistema se descompone en un conjunto de objetos que se comunican entre ellos.

2. *Un modelo de flujo de datos*: El sistema se descompone en módulos funcionales que afectan entradas de datos y las transforman de alguna manera en datos de salida.

3.2. Especificación Abstracta

Se entiende por **especificación** a la determinación, explicación o detalle de las características o cualidades de una cosa y por **abstracción** a la habilidad de considerar aisladamente las

cualidades esenciales de un objeto, o el mismo objeto en su pura esencia o noción; se identifican los aspectos importantes, se examinan selectivamente ciertos aspectos de un problema [20].

La especificación abstracta en el contexto del diseño de software se puede definir como la especificación de las características, de sus servicios y las restricciones bajo las cuales opera, delimitando su alcance, estableciendo las interfaces con otros sistemas e identificando a los usuarios representativos.

La especificación abstracta consiste en documentar cada uno de los subsistemas, describiendo su funcionalidad y especificando bajo que restricciones opera el subsistema, sin poner atención en los detalles del subsistema, con el objetivo de obtener:

- La estructura genera del proceso.
- El comportamiento general del proceso.
- La estructura de la interrelación de los procesos.
- El control general.
- Interrupciones y excepciones.

Una de las principales problemáticas que existe es que la especificación de la funcionalidad de los sistemas se hace con lenguaje natural. Esto genera problemas en la definición de los subsistemas, por lo que se ha visto la necesidad de ocupar estrategias que nos orientan a usar lenguajes formales. Sin embargo, esto también tiene desventajas ddado que los lenguajes formales para la especificación son complejos de entender y sólo pueden ser interpretados por los expertos. La especificación abstracta busca que la especificación pueda ser entendida, por: los experto del modelo de negocios, programadores y en general todos los usuarios y desarrolladores que participan de alguna forma en el sistema, la especificación abstracta parte de que cada bloque del sistema representa un cierto subsistema o modulo, y este realiza cierta(s) tarea(s) relacionadas con el modelo de negocios. La especificación abstracta no explica como los módulos

serán implementados, ni como se diseñan las tareas, sino que especifica la funcionalidad de los modulos.

Con la especificación abstracta queremos entender el funcionamiento del sistema, mas no los detalles de como se realizan las tareas de cada subsistema, se quiere entender que se debe prever, con el objetivo de hacer cambios (si son necesarios). Esto ayuda a disminuir los riesgos en el diseño de un sistema, por que desde el inicio se debe entender como se va a construir un sistema y de que subsistemas se conforma.

3.3. Diseño de Interfaz

Una interfaz tiene que ver con los dispositivos de software o hardware que manejan entradas y salidas, por lo que la forma de manejar las entradas y las salidas deben estar bien definidas.

Existen muchos tipos de interfaces pero nos centraremos en dos tipos específicos:

1. El diseño de las interfaces entre los componentes y/o subsistemas.

2. El diseño de las interfaces entre el hombre y la computadora.

3.3.1. Interfaz entre Subsistemas

Los sistemas grandes se descomponen en subsistemas que se desarrollan de forma independiente. Éstos utilizan otros subsistemas, por lo que una parte esencial del proceso de especificación consiste en definir como interactúan dichos subsistemas.

Al mecanismo que ofrecen los subsistemas para interactuar entre ellos mismos se le denomina interfaz. La interfaz es el medio por el cual se establece la comunicación del subsistema con otros subsistemas. La interfaz es importante por que permite modificar al subsistema sin la necesidad que el entorno al subsistema se entere de los cambios.

Para evitar ambigüedades se pueden definir formalmente usando un lenguaje como el IDL(pos sus siglas en inglés Interface Defnition Language).

3.3.2. Interfaz entre Usuario y software

La etapa del diseño de las interfaces del software con el usuario es básica, dado que a través de esta se le mostrará al usuario como interactuará con el sistema. Sin embargo es importante cuidar aspectos tales como [12]:

1. *Dar el control al usuario*: Que el sistema reaccione ante las necesidades del usuario y que lo ayude a hacer las cosas. Es importante considerar los siguientes principios para la creación de una interfaz:

    - Definir los modos de interacción de manera que no obligue a que el usuario realice acciones innecesarias y no deseadas.
    - Permitir que las interacciones del usuario se puedan interrumpir y deshacer.
    - Disminuir la interacción a medida que avanza el nivel de conocimientos y permitir personalizar las interacción.
    - Ocultar al usuario los problemas técnicos.

2. *Reducir la carga de memoria del usuario*: Cuánto más tenga que recordar un usuario es más propenso a errores, por lo que hay que considerar:

    - Reducir la demanda de memoria a corto plazo. Hay que recordarle al usuario sus acciones y resultados anteriores, mediante claves visuales.
    - Establecer valores por defecto útiles.
    - El formato visual de la interfaz deberá ser lo más apegado al mundo real.
    - Desglosar la información de forma progresiva.

3. *Construir una interfaz uniforme*: La interfaz deberá adquirir y presentar la información de forma uniforme, lo que implica:

- Que la información visual este organizada de acuerdo con el diseño estándar que se presenta en todas las presentaciones de pantalla.
- Que todos los mecanismos de entrada estén limitados y que se utilicen de forma uniforme.
- Los mecanismos para ir de tarea en tarea se hayan definido e implementado de forma uniforme.
- Mantener la consistencia en toda la familia de aplicaciones

La creación de la interfaz de software es un proceso iterativo, incluye las siguientes etapas [19]:

1. Análisis y modelado de usuarios tareas y entornos.
2. Diseño de la interfaz.
3. Implementación de la interfaz.
4. Validación de la interfaz.

3.4. Diseño de Componentes

El diseño de componentes consiste en convertir el modelado de datos, interfaces y arquitectura en un software operacional. Cada subsistema que describimos en la arquitectura puede verse como un componente dado que tienen una funcionalidad especifica, con sus interfaces definidas para interactuar con otros subsistemas o usuarios. El objetivo del *diseño de componentes* es especificar como se construirá el componente de forma clara y especifica, en caso de no contar con él.

Cuando se inicia la etapa de *diseño de componentes* contamos con la información necesaria de que va a hacer el componente. Sin embargo aún no sabemos cómo lo hará, ni como lo construiremos, ni como lo dividiremos para su creación. Por lo cual en el diseño de componentes

también es necesario definir las interfaces del componente para establecer, como se comunicara con otros componentes.

Durante la etapa del diseño de componentes, se espera obtener como resultado como se deben construir los componentes del sistema, y no solo que hacen los componentes. Los componentes con los que se conformará un sistema pueden ser diseñados de múltiples formas y con diferentes paradigmas y técnicas de programación, sin embargo solo citaremos:

- El diseño funcional.
- El diseño orientado a objetos.

3.4.1. Diseño Funcional

El diseño funcional ha sido una práctica informal desde los inicios de la programación. La idea es descomponer las funciones en subrutinas que realmente son funciones mas sencillas, en donde se tratan de esconder los detalles de los algoritmos a través de funciones, sin embargo lo que no puede esconder es la modificación que sufren tanto los datos como el estado del sistema.

El diseño funcional trabaja apoyado de:

- *Diagramas de flujo de datos*: Estos muestran como los datos pasan a través del sistema y se transforman por cada función del sistema.
- *Descomposición estructural*: Modela como la función es descompuesta en subfunciones, usando una estructura gráfica.
- *Descripción de los detalles del sistema*: Describe las entidades en el diseño y sus interfaces. Esta descripción puede ser registrada en un diccionario de datos. La estructura de control del diseño se describe usando un lenguaje de descripción de programas, el cual debe incluir expresiones condicionales y construcciones de ciclos (loops).

3.4.2. Diagramas de flujo de Datos

Los diagramas de flujo de datos, son una representación gráfica, que permite analizar y definir las entradas, procedimientos y salidas de la información, en la organización bajo estudio.

El diagrama de flujo de datos (DFD) tiene por objetivo representar gráficamente el sistema a nivel lógico y conceptual, ilustrando los componentes esenciales de un proceso y la forma en que estos interactúan.

Los diagrama de flujo de datos permiten:

- Representar gráficamente los límites del sistema en estudio.
- Mostrar el movimiento de los datos y la transformación de los mismos a través del sistema.
- Facilitar el mantenimiento del sistema.

Los componentes de un diagrama típico de flujo de datos son:

- Proceso.
- Flujo.
- Almacén.
- Terminador.

3.4.2.1. Proceso

Es una actividad que transforma o manipula datos y representa los procedimientos utilizados para transformar los datos.

Son importantes los siguientes aspectos para su construcción:

- Un proceso no es origen ni final de los datos, sólo lugar de transformación de los mismos.
- Un proceso puede transformar un dato en varios datos.

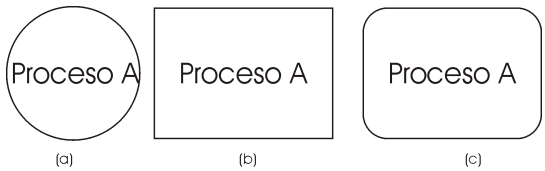


Figura 3.1: Símbolos para definir un proceso en un DFD.

- Es necesario un proceso que sirva como intermediario entre una entidad externa y un almacén de datos.

El proceso se representa gráficamente como un círculo, (como se muestra en figura 3.1(a)). Algunos prefieren usar un rectángulo, como se muestra en la figura 3.1(b). y otros prefieren usar un óvalo o un rectángulo con esquinas redondeadas, como se muestra en la figura 3.1(c). Las diferencias entre estas tres formas son puramente cosméticas, aunque obviamente es importante usar la misma forma de manera consistente para representar todas las funciones de un sistema.

Un proceso se nombra o describe con una sola palabra, frase u oración sencilla. Un buen nombre para un proceso generalmente consiste en una frase verbo-objeto tal como validar entradas o calcular impuesto. En algunos casos, el proceso contendrá el nombre de una persona o un grupo (por ejemplo, un departamento o una división de una organización), o de una computadora o un aparato mecánico.

3.4.2.2. Entidades Externas

Estas representan entidades ajenos a la aplicación, pero que aportan o reciben información de la misma. Generalmente se ven como clases lógicas de cosas o de personas, las cuales representan una fuente o destino de transacciones, con las que el sistema se comunica. Como por ejemplo clientes, empleados, proveedores. También pueden ser una fuente o destino específico, como por ejemplo Departamento Contable.

Dado que las entidades externas están fuera de los limites del sistema, no se toman en cuenta las transformaciones o procesos que sufren los datos dentro de ellas, es decir los datos están

Figura 3.2: Símbolos para definir una Entidad Externa en un DFD.

fuera del control del sistema.

Al momento de identificar las entidades externas, se debe de tener en cuenta lo siguiente:

- Las entidades externas representan personas, organizaciones o sistemas que no pertenecen al sistema.
- En el caso que las entidades externas se comunicasen entre si, esto no se contemplaría en el diagrama, por estar fuera del ámbito del sistema.
- Pueden aparecer en los distintos niveles de DFD.
- Pueden aparecer varias veces en un mismo diagrama, para evitar entrecruzamiento de líneas.
- Suministra información acerca de la conexión del sistema con el mundo exterior.

Se representan mediante un rectángulo como figura 5.15

3.4.2.3. Almacén de datos

Un almacén de datos representa un depósito de información dentro del sistema, un archivo lógico, en donde se agregan o de donde se extraen datos. Para identificar los lugares de almacenamiento de datos, se debe de tener en cuenta lo siguiente :

- Los almacenes de datos representan la información en reposo.
- Los almacenes de datos no puede crear, destruir y transformar datos.
- No pueden estar comunicado directamente con otro almacén o entidad externa.

Figura 3.3: Símbolos para definir un flujo de datos en un DFD.

- El flujo de datos (entrada o salida) no lleva nombre cuando incide sobre su contenido completo.
- El almacén de datos aparecer por vez primera, en aquel nivel en que sea accedido por 2 o más procesos y en modo lectura y/o escritura.
- No debe de estar referido al entorno físico.
- No se representa la clave de acceso a la información sino sólo a la operación que se realiza.

Se denota por dos líneas paralelas, como lo muestra la figura 3.3

3.4.2.4. Flujos de datos

Los flujos de datos establecen la comunicación entre procesos, almacenes y entidades externas y llevan información para esos objetos.

Para elaborar los flujos de datos deber de tenerse en cuenta que:

- Se deben de conocer los flujo de información en la estructura.
- Los datos no pueden ser creados ni destruidos por un flujo de datos.
- Sirve para conectar el resto de los componentes del DFD.
- No es un activador de procesos.

Un flujo se representa gráficamente por medio de una flecha que entra o sale de un proceso; un ejemplo se muestra en la figura 3.4. El flujo se usa para describir el movimiento de bloques o paquetes de información de una parte del sistema a otra.

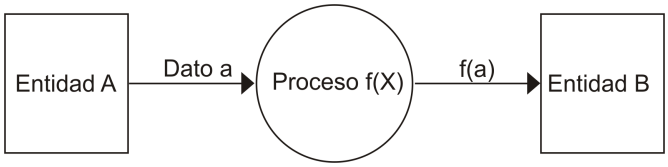


Figura 3.4: Símbolos para definir un flujo de datos en un DFD.

3.4.2.5. Pasos para la creación de un Diagrama de Flujo de Datos (DFD)

A continuación enumeraremos los pasos básicos para la creación de un DFD.

1. Primero se deberán identificar las entidades externas ya que ello implica definir los límites del sistema
2. Se deberán elegir nombres con significado tanto para procesos como también para flujos de datos, almacenes y entidades externas. Si es posible a partir del vocabulario del usuario evitando terminologías técnicas.
3. Identificar el papel del proceso del sistema, no quien lo realiza.
4. Numerar los procesos, mediante un esquema de numeración consistente que implique, para los lectores del DFD, una cierta secuencia de ejecución.
5. Se deberán, en la medida de lo posible, evitar los DFD excesivamente complejos. Deberán ser comprensibles, digeribles y agradables a la vista sin demasiados elementos.
6. Todos los elementos se relacionan entre sí a través de flujos de datos.
7. Procesos, se relacionarán con:
   - Almacenes
   - Entidades externas
   - Otros procesos
   - Deberán tener al menos una Entrada y una Salida, no son manantiales de datos.

8. Almacenes, se relacionarán solamente con Procesos.

9. Entidades Externas, se relacionarán solamente con Procesos.

10. En todos los niveles del Diagrama de Flujo de Datos deberá haber igual cantidad de Entradas y de Salidas.

11. Niveles del DFD:

    a) Nivel de Partida, Diagrama de Contexto:

    - No existirán almacenes o archivos.
    - Se representarán las entidades externas que son fuente y destino de los datos.
    - El sistema será representado como un proceso simple.
    - Se dibujarán sólo los flujos de datos de comunicación exterior-sistema.

    b) Nivel 1 y subsiguientes:

    - Deberá haber igual cantidad de archivos. Aunque podrá existir mayor cantidad de almacenamientos en el nivel 2 debido a la explosión de algún proceso.
    - En el último nivel, cada proceso realizará una función específica y concreta.

12. Cada proceso en el DFD de alto nivel de un sistema puede ser ”explotado”para convertirse en un DFD en si mismo.

13. Cada proceso en el nivel inferior deberá estar relacionado, inversamente, con el proceso del nivel superior. Es decir que, cada proceso padre que se detalla en el DFD, ha de estar balanceado. La regla del balanceo consiste en que cada proceso debe tener exactamente los mismos datos de entrada/salida netos que el DFD hijo.

14. Los flujos de datos pueden descomponerse en la explosión del proceso en un DFD hijo.

3.4.3. Descomposición Estructural

La descomposición estructural se utiliza para modelar la estructura del sistema. Este modelo estructural muestra como una función es realizada por un número de funciones que son invocadas. Esta descomposición es representada gráficamente con rectángulos que representan funciones y flechas que indican el flujo de la información de entrada y salida, este modelo es jerárquico, el de más arriba invoca a las funciones de abajo. Como se observa en la figura 3.5 $F(x)$ realiza su tarea puede invocando a $f_1(z)$, $f_2(y)$ y a $f_3(w)$, sin embargo como se observa en la figura 3.5 $f_1(z)$ invoca a $f_{11}(z')$, $f_{12}(y')$ y $f_{1n}(w')$ para realizar su trabajo.

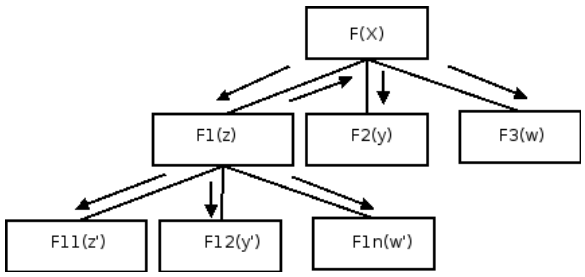


Figura 3.5: Estructuras de los diagramas de flujo

Para realizar la descomposición estructural es necesario considerar:

- Esto es muy útil para sistemas que manejan entradas de datos, procesamiento de datos, validación y chequeo de datos .
- Si se desea validar los datos debe realizarse con funciones subordinadas, las salidas para dispositivos así como el formateo de los datos también debe realizarse con funciones subordinadas controladas por una función central.
- La funciones definidas en la parte superior coordinan y controlan a las que se encuentran en niveles inferiores.
- El objetivo de este tipo de descomposición es lograr bajo acoplamiento y alta cohesión.
- Cada función debe de tener entre 2 y 7 funciones subordinadas como máximo.

Por lo tanto para lograr la descomposición estructural es necesario seguir los siguientes pasos:

- Identificar los procesos de transformación del sistema.
- Identificar la transformación de las entradas.
- Identificar la transformación de las salidas.

3.4.3.1. Descripción de los detalles del sistema

En esta etapa lo que se busca es poder organizar y especificar las funciones del sistema para saber que es lo que hacen cada función, sus entradas y salidas. También se desea obtener la descripción y especificación de las entidades. Esto ayuda al proceso de diseño a identificar los defectos provocados por la descomposición funcional.

Por lo que una estrategia para manejar esta etapa es ocupar un diccionario de datos. En este diccionario se registran las funciones y las entidades, permitiendo el mantenimiento de los nombres de las funciones y su descripción. Esto reduce la posibilidad de usar nombres ya provistos y produce un diseño que puede ser leíble permitiendo entender que es lo que hizo el diseñador.

Los detalles de los datos que se registran en los diccionarios de datos puede variar ya que pueden hacerse descripciones largas o cortas o se pueden utilizar lenguajes formales para su especificación.

Las funciones pueden ser especificadas usando estructuras gráficas ó con diagramas de flujo. Esto es sencillo dado que se ocupan cajas que indican un paso del proceso, un rombo representa una condición lógica y las flechas indican el flujo de control. La figura 3.6 ilustra tres condiciones estructuradas.

Estas especificaciones se pueden realizar ocupando las siguientes construcciones:

- *construcción secuencial* implementa los pasos del proceso esenciales para la especificación de cualquier algoritmo.

- *construcción condicional* proporciona las funciones para procesos seleccionados a partir de una condición lógica y la
- *construcción repetitiva* proporciona los bucles.

Estas construcciones se propusieron para restringir el diseño procedural del software a un número reducido de operaciones predecibles, con el objetivo de comprenderlo, comprobarlo y mantenerlo. La utilización de un número limitado de construcciones lógicas contribuye a un proceso de comprensión humana.

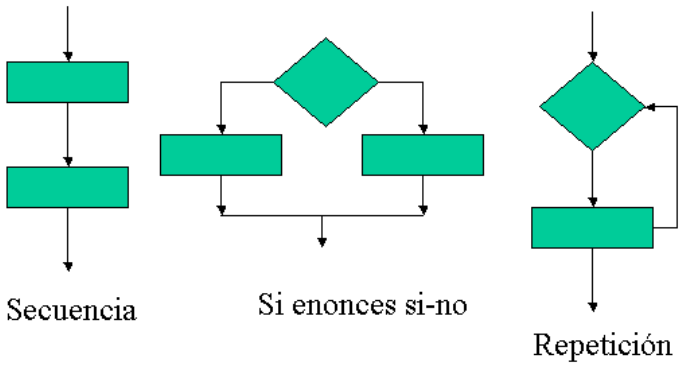


Figura 3.6: Estructuras de los diagramas de flujo

Las construcciones estructuradas pueden anidarse unas en otras, anidando construcciones. Se puede desarrollar un esquema lógico complejo, destacando que, un bloque puede representar un módulo, logrando por lo tanto la estratificación procedural que con lleva a la estructura del programa [19].

3.4.4. Diseño Orientado a Objetos

El diseño orientado a objetos (DOO) es una estrategia en la cual el diseñador del sistema piensa en términos de objetos, en lugar de operaciones o funciones. El DOO tiene como base: la abstracción, el ocultamiento de información, la independencia funcional y la modularidad, el DOO comprende el diseño de clases y sus relaciones.

Los sistemas Orientados a Objetos, son fáciles de mantener, ya que los objetos son independientes. Los objetos pueden ser entendidos y modificados como entidades auto-contenidas por lo que se les puede agregar funcionalidad sin que afecte a otros sistemas. Los objetos son reutilizables para algunos sistemas y puede haber un mapeo entre las entidades del mundo real y los objetos del sistema.

3.4.4.1. Características principales del Diseño Orientado a Objeto

- Los objetos son abstracciones del mundo real o entidades del sistema que se administran entre ellas mismas.
- Los objetos son independientes y encapsulan el estado y la representación de información.
- La funcionalidad del sistema se expresa en términos de servicios de los objetos.
- Las áreas de datos compartidas son eliminadas. Por lo que los objetos se comunican mediante mensajes que pueden o no incluir parámetros.
- Los objetos pueden estar distribuidos y pueden ejecutarse en forma secuencial o en paralelo.

3.4.4.2. Desarrollo Orientado a Objetos

El análisis, diseño y programación orientada a objetos son parte de del desarrollo orientado a objetos sin embargo abarcan diferentes áreas del desarrollo orientado a objetos.

- ***El análisis orientado a objetos:*** Trata las estrategias para obtener el dominio del problema (entender el modelo de negocios) desde la perspectiva de los objetos, identificando los principales objetos que seran mapeados al sistema.
- ***El Diseño Orientado a Objetos:*** Con base en los requerimientos y el modelo de negocios se estructura como se implementará el sistema.

- ***La programación orientada a objetos:*** Es la implementación del diseño orientado a objetos, ocupando algún lenguaje orientado a objetos.

3.4.4.3. Método de Diseño Orientado a Objetos

Algunos métodos que han sido propuestos son los de Coad Yourdon, 1990; Robinson, 1992; Jacobson, 1993; Booch 1994; Graham, 1994; RUP, 1998. Mas a delante se describe el de RUP sin embargo revisaremos lo que tienen en común:

- La identificación de objetos, sus atributos y servicios.
- La organización de objetos dentro de una agregación jerárquica la cual muestra como los objetos son parte de otros objetos.
- La construcción de descripciones dinámicas de objetos que muestran como ciertos objetos son usados por otros objetos suministrandoles cierto servicio.
- La especificación de interfaces de objetos.

Las actividades anteriormente mencionadas, son fundamentales en el desarrollo orientado a objetos(OO), estas se combinan y enriquecen para obtener un mejor desarrollo OO, basado basicamente en el diseño de los objetos, y sus servicios y operaciones de estos.

3.4.4.4. Objeto

Los objetos se han definido como la instancia de una clase, un objeto es una entidad que tiene un estado y un conjunto definido de operaciones, que operan sobre el estado del objeto; El estado de un objeto es determinado por los valores que asumen el conjunto de atributos del objeto. Las operaciones asociadas con el objeto proveen servicios a otros objetos (clientes) que requieren estos servicios cuando necesitan realizar alguna actividad de cómputo. Los objetos se crean de acuerdo a una definición de la clase objeto. La definición de la clase objeto sirve como

plantilla para los objetos. Esta incluye las declaraciones de todos los atributos y servicios, los cuales deben estar asociados con un objeto de esta clase.

El proceso de diseño orientado a objetos se refiere al diseño de las clases objetos, ya que cuando el diseño se implanta se requiere la creación de objetos que son creados usando la definición de la clase.

Comunicación entre objetos. Los objetos se comunican mediante paso de mensajes, por peticiones de servicios entre objetos, estos mensajes pueden ser de texto; en la practica, los mensajes se implementan a menudo mediante llamadas de procedimientos, estos procedimientos permiten el intercambio de parámetros, y la modificación a ciertos parámetros que provocan los cambios de estado de los objetos.

3.4.5. Conceptos Importantes en el Diseño Orientado a Objetos (DOO)

Cuando se hace DOO es necesario considerar 3 características propias de los objetos que son: Herencia, Polimorfismo y Encapsulamiento que permiten, un diseño más eficiente desde el punto de vista mantenimiento y reutilización. También revisaremos los conceptos de Cohesión y Acoplamiento que mejoran el mantenimiento de un sistema.

- **Herencia**: El crear una clase con todo el trabajo que esto implica y el crear una nueva clase con una *funcionalidad similar*, es algo poco deseado. Por lo que lo ideal sería hacer uso de una clase ya existente, clonarla y después hacer al “clon”las adiciones y modificaciones que sean necesarias. Esto se puede lograr mediante la herencia. La clase base es a la que se le denomina super-clase o clase padre, y al “clon”se le denomina clase derivada, clase heredada, subclase o clase hijo, como se ve en la figura 3.7 usando notación UML.

  Cuando los objetos son creados ellos heredan los atributos y operaciones de la super-clase. Las clases objeto es en si mismo, un objeto, así que hereda sus atributos de alguna otra clase(su super-clase).

  Un tipo de clase define los límites de un conjunto de objetos, que puede tener relación con

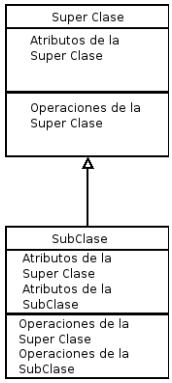


Figura 3.7: Ejemplo de Herencia

otros tipo clases. Dos tipos de clases pueden tener características y comportamientos en común, pero una puede contener más características que otra y también puede manipular más mensajes (o gestionarlos de manera distinta). La herencia expresa esta semejanza entre tipos de clases haciendo uso del concepto de: clase base y clase derivada. Una clase base define una serie de características y comportamiento que puede compartir con las clases que de él derivan. A las clases derivadas además se les incrementa su funcionalidad (se especializan) a partir de la clase base. Veamos un ejemplo sencillo en la figura 3.8

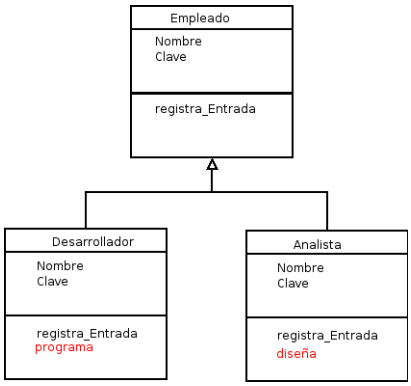


Figura 3.8: Ocupando Herencia para los empleados de una empresa

donde se describe el diagrama de clases de los empleados de una empresa de creación de software, todos los que laboran en esta empresa son trabajadores(definimos una clase

base), por lo tanto todos tienen nombre, clave, y todos registran su entrada. Sin embargo existen diferentes tipos de empleados en la empresa algunos son desarrolladores (clase derivada) y otros son analistas (clase derivada), por lo tanto ofrecen diferentes servicios a la empresa sin perder de vista que son trabajadores, los analistas diseñan mientras los desarrolladores programan (especialización de las clases derivadas), en la super-clase tenemos los atributos y operaciones generales que tendrán todas las clases que hereden y en las subclase tenemos las características de la super-clase y sus características propias, que lo especializan.

De esta forma la herencia nos permite hacer una reutilización rápida de lo que son algunos clases ya diseñadas haciendo especializaciones.

Dicho con otras palabras la herencia permite que una clase obtenga las propiedades y operaciones de una super-clase, y puede especializarse tomando como base las características de la super-clase.

- **Polimorfismo**: La palabra polimorfismo tiene que ver con las múltiples formas que puede tener cierta cosa. Por tal razón el polimorfismo en el DOO ofrece mecanismos para resolver una petición de multiples formas.

  Usualmente los métodos tienen definidos sus parámetros, por ejemplo si deseamos crear un objeto línea sabemos que se puede hacer proporcionando dos puntos, pero también se podría hacer dando una pendiente y un punto, realmente esto se puede realizar de múltiples formas, si por alguna razón no se supieran que parámetros se van a recibir, sería muy complejo pensar en diferentes nombres para la creación de una clase que haga objetos tipo línea.

  Por lo que en paradigma orientado a objetos existe un mecanismo denominado polimorfismo, que permite definir dos o más métodos diferentes utilizando el mismo nombre.

  El criterio que utilizan los compiladores para saber que método invocar va a depender de los parámetros que tenga el método.

En resumen, el polimorfismo permite que un número de operaciones diferentes tengan el mismo nombre, esto reduce el acoplamiento entre los objetos haciendo a cada uno más independiente.

- **Encapsulamiento**: Es un mecanismo para ocultar los detalles internos de un objeto de los demás objetos. En otras palabras los atributos (datos) de un objeto no pueden ser modificados o accedidos directamente por otro objeto, para acceder a los datos de un objeto se debe hacer mediante sus métodos o interfaces que el objeto ofrece. El encapsulamiento oculta los detalles de como trabaja el objeto internamente, se puede ver como una caja negra. En la figura 3.9 podemos observar que el acceso a los datos de un objeto solo debe

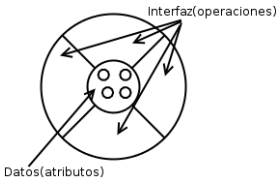


Figura 3.9: Estructuras de los diagramas de flujo

de se a través de sus metodos, todo el detalle, al estar encapsulado, es desconocido por el resto de la aplicación, limitando el impacto de cualquier cambio interno del objeto. Obviamente, cualquier cambio en la interfaz del objeto podrá afectar potencialmente al resto de la aplicación. Sin embargo, el porcentaje de código dedicado a la interfaz es, en general, mucho menor que el total de líneas de código utilizadas para datos e implementación de las operaciones. Por lo que se reduce la complejidad del sistema, protegiendo los objetos frente a posibles errores y logrando extensiones futuras de estos.

- **Cohesión** Un módulo cohesivo lleva a cabo una sola tarea dentro de un procedimiento de software, lo cual requiere poca interacción con los procedimientos que se llevan a cabo en otra parte del programa, dicho de manera sencilla un módulo cohesivo deberá (idealmente) hacer una sola cosa [19], entre más especifica es el módulo decimos que la cohesion es más alta y siempre buscamos cohesion alta.

- **Acoplamiento** El acoplamiento depende de la complejidad de interconexiones entre los módulos, el punto donde se realiza una entrada o referencia a un módulo, y los datos que pasan a través de la interfaz. El diseño de software, intenta obtener el acoplamiento bajo, Una conectividad sencilla entre los módulos da como resultado un software más fácil de entender y menos propenso a tener un efecto ola.

3.5. Diseño de Algoritmos y Estructura de Datos

Se crea un algoritmo par implementar la especificación para cada operación. En muchas ocasiones, el algoritmo es una secuencia computacional o procedural, que pude ser implementada como un módulo de software auto contenido, sin embargo, si la especificación de la operación es compleja, será necesario, modularizar la operación con las técnicas convencionales de diseño.

Las estructuras de datos se diseñan al mismo tiempo que los algoritmos, ya que las operaciones manipulan los atributos de una clase, el diseño de estructuras de datos, que reflejan mejor los atributos, tendrán un fuerte sentido en el diseño algorítmico de las operaciones correspondiente.

Aunque existen muchos tipos diferentes de operaciones, normalmente se pueden dividir en tres grandes categorías:

1. Operaciones que manipulan los datos

2. Operaciones que ejecutan cálculos

3. Operaciones que monitorean al objeto para la ocurrencia de un suceso controlado.

Estos diseños pueden ser mejorados si utilizamos patrones de diseño que explicaremos posteriormente.

3.6. Características de calidad

Hoy en día es necesario tener software de calidad, se dice que cierto software tiene calidad si cumple 6 criterios descritos en la norma ISO 9126 que a continuación mencionaremos:

- *Funcionalidad.*
- *Fiabilidad.*
- *Usabilidad.*
- *Eficiencia.*
- *Mantenibilidad.*
- *Portabilidad.*

3.6.1. Funcionalidad

Es el conjunto de atributos que permiten que un sistema provea ciertas propiedades y funciones especificas en otras palabras es el grado en que las necesidades asumidas o descritas se satisfacen. Estas se subdividen de la siguiente forma:

- Adecuación.
- Corrección.
- Interoperatibilidad.
- Seguridad.
- Conformidad.

3.6.2. Fiabilidad

Es el grado en que el sistema responde bajo las condiciones definidas durante un intervalo de tiempo dado. Estas se subdividen de la siguiente forma:

- Madurez.
- Tolerancia a Fallos.
- Recuperabilidad.

3.6.3. Usabilidad

La usabilidad determina el grado de esfuerzo para que un usuario aprenda y use cierto software, que tan productivos son los usuarios que trabajan con ese sistema y cuánta ayuda necesitarán. Estas se subdividen de la siguiente forma:

- Aprendibilidad.
- Comprensibilidad.
- Operabilidad.
- Atractividad.

3.6.4. Eficiencia

ES el conjunto de características que determinan la relación entre el nivel de rendimiento del software y el número de recursos usados, bajo ciertas condiciones dadas. Estas se subdividen de la la siguiente forma:

- Comportamiento Temporal.
- Utilización de recursos.

3.6.5. Mantenibilidad

Es el esfuerzo requerido para implementar cambios. Estas se subdividen de la la siguiente forma:

- Analizabilidad.
- Cambiabilidad.
- Estabilidad.
- Facilidad de Prueba.

3.6.6. Portabilidad

Es el conjunto de características que determinan la capacidad del software para ser transferido de un entorno de operación a otro. Este se subdividen de la la siguiente forma:

- Adaptabilidad.
- Instalabilidad.
- coexistencia.
- Reemplazabilidad.

Capítulo 4

Proceso Unificado de Rational

Este capítulo se dividirá en dos secciones. La primera sección tiene como objetivo dar una introducción a lo que es el Proceso Unificado de Rational (Rational Unified Process RUP), sus estrategias de trabajo y su evolución. La segunda sección de este capítulo explicará el proceso del RUP con base en su arquitectura.

4.1. Introducción

El RUP es un proceso de desarrollo de software que describe un conjunto de actividades para transformar los requerimientos del cliente en un sistema de software. El RUP es un proceso de Ingeniería de software que tiene como objetivo el asignar tareas y responsabilidades, para producir software de alta calidad, buscando satisfacer las necesidades de los clientes, ajustándose al presupuesto y a los tiempos estimados. El RUP ofrece diferentes procesos, que brindan un marco de trabajo, adaptable y extendible a las necesidades de cada organización.

El RUP captura las mejores prácticas actuales de desarrollo de software, por lo que se basa en las buenas prácticas que ocuparon diferentes equipos de desarrollo de distintas empresas. Estos equipos de desarrollo fueron seleccionados por su éxito en el desarrollo de software. Después de revisar un gran número de proyectos, el RUP adoptó de forma general las siguientes 6 prácticas

de desarrollo de software.

- Desarrollo iterativo de software.
- Manejo de requerimientos.
- Uso de una arquitectura basada en componentes.
- Modelo de software visual.
- Verificación continua de la calidad de software.
- Control de cambios para el software.

El RUP se basa en el modelo de espiral de Barry Boehm quien propone identificar los riesgos del proyecto en una etapa temprana de su ciclo de vida, cuando es posible atacarlos y reaccionar eficazmente. El RUP no usa un modelo lineal para el desarrollo de software, porque este incrementa los riesgos conforme va pasando el tiempo. Sin embargo, muchos desarrollos de software se han realizado con el proceso de cascada que es un modelo lineal que comprende las siguientes etapas: análisis de requerimientos, diseño, codificación y pruebas de módulos, pruebas de subsistemas y finalmente pruebas del sistema. El principal problema del modelo de cascada, como se mencionó anteriormente, es el incremento de los riesgos del proyecto conforme va pasando el tiempó, tal com se observa en la figura 4.1. Por tal razón el RUP adopto un modelo iterativo.

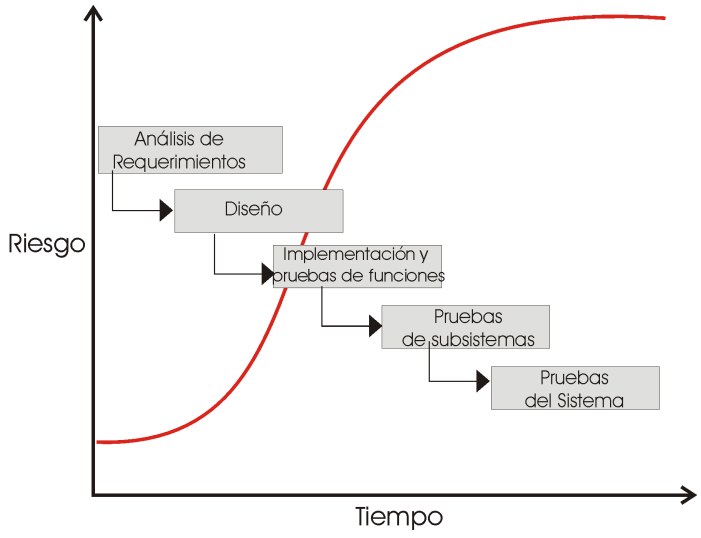


Figura 4.1: Modelo de cascada

El RUP utiliza una estrategia iterativa como se observa en la figura 4.2, donde cada iteración incluye diferentes disciplinas de desarrollo.

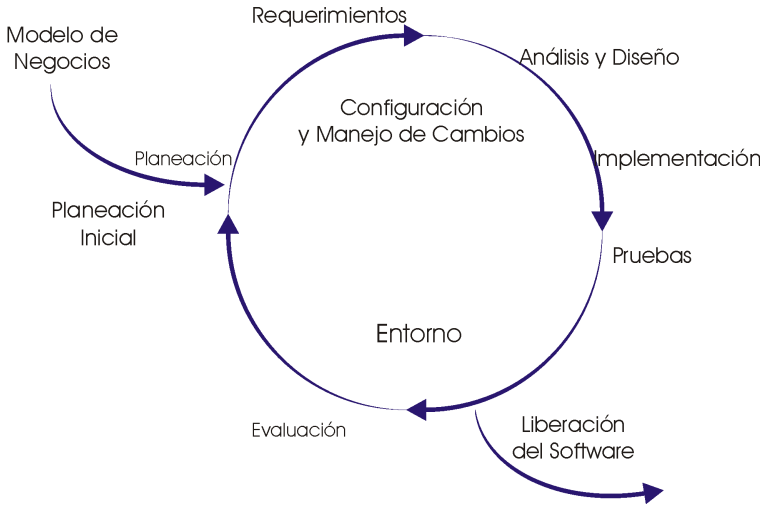


Figura 4.2: El proceso iterativo del RUP

4.1.1. Evolución del RUP

El RUP a madurado a través del tiempo recopilando la experiencia de diferentes personas y compañías. En la figura 4.3 se presenta un esquema de esta evolución.

4.2. Arquitectura del RUP

La arquitectura del RUP se divide en dos partes, la estructura dinámica y la estructura estática (tal como se observa en la figura 4.4).

- Estructura Dinámica: Trata con el ciclo de vida o con la dimensión del tiempo. Se representa por la dimensión horizontal, (como se observa en la figura 4.4) y tiene 4 fases que son: *inicio, elaboración, construcción y transición*. Cada fase tiene un punto de revisión y contiene una o más iteraciones, las cuales se enfocan en producir la información técnica necesaria para obtener los objetivos de la fase.

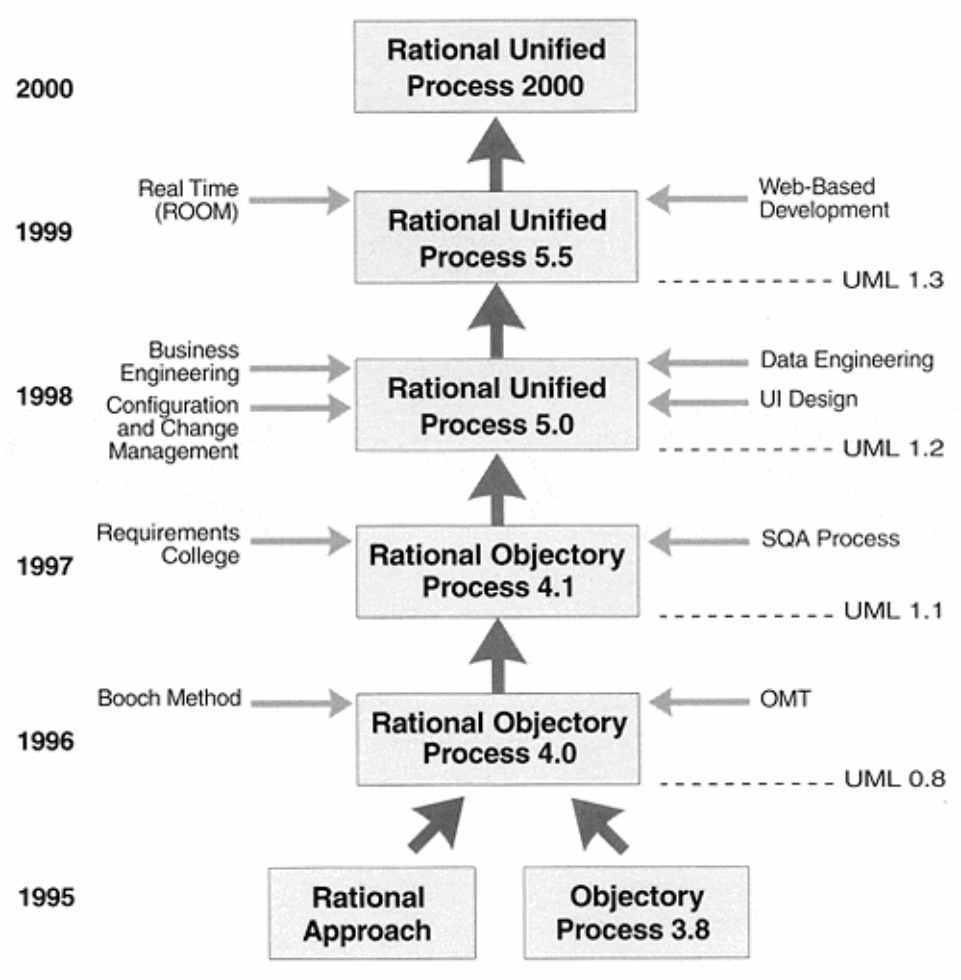


Figura 4.3: Historia del RUP

- Estructura estática: Trabaja con actividades, flujos de trabajo, artefacto de software y roles. La estructura estática está ilustrada por la dimensión vertical en la figura 4.4.

4.3. Estructura Dinámica

La estructura dinámica del RUP contempla 4 fases, cada fase tiene objetivos propios y para cumplirlos la fase desarrolla todos y cada uno de los pasos indicados en la estructura estática (la cual se explicara más adelante), por lo que cada fase itera sobre la estructura estática, tantas

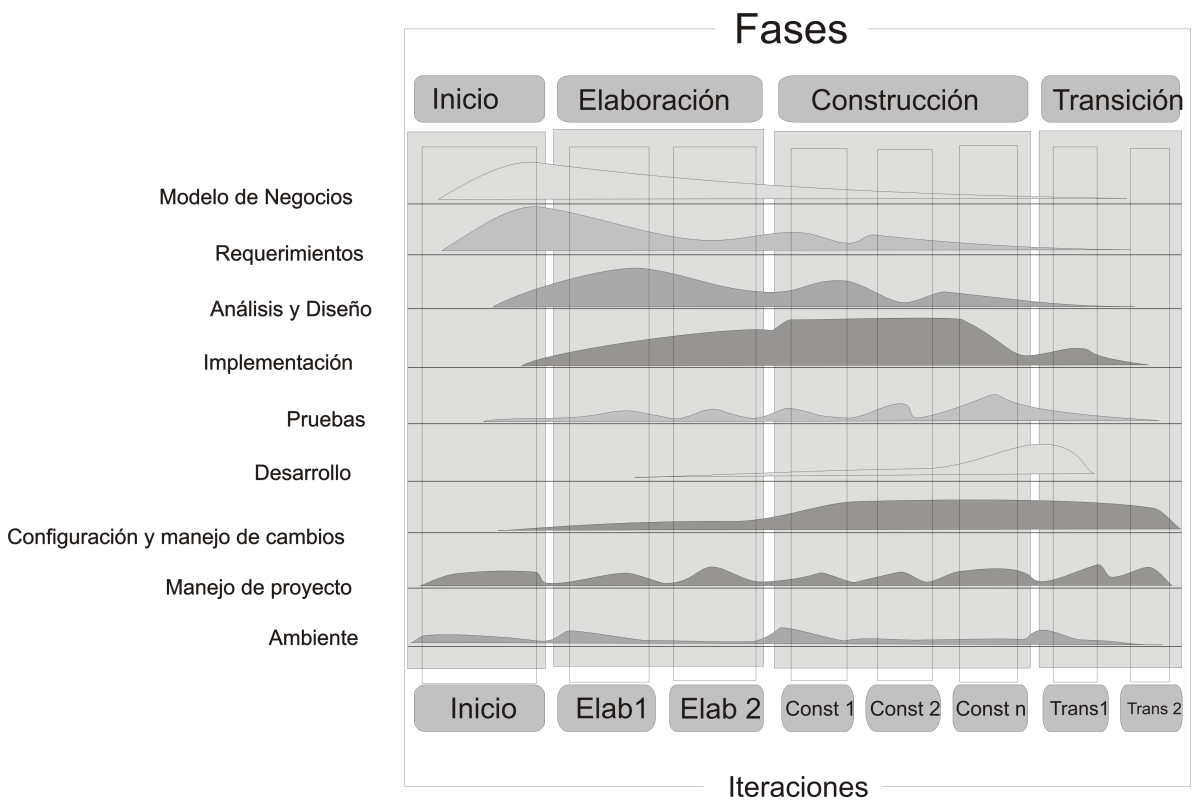


Figura 4.4: Arquitectura del RUP

veces como se haya planeado. Las fases de la estructura dinámica son:

- Inicio (Inception).
- Elaboración (Elaboration).
- Construcción (construction).
- Transición (transition).

El RUP determina que al terminar la fase de transición, se genera un sistema ejecutable. A continuación explicaremos las actividades que se realizan en cada fase y cuales son sus objetivos, así como los artefactos de software que se generan en cada fase.

4.3.1. Iniciación (Inception)

Cuando se toma la decisión de hacer cierto software, lo mas importante es analizar el problema para entender que se quiere hacer. Se debe estimar el costo y tiempo que tomará el realizarlo. A esta primera etapa en el proceso de RUP se le llama Iniciación y tiene los siguientes 5 objetivos:

1. Entender que es lo que se desea construir con el fin de determinar la visión, el alcance del sistema y sus limites. Así mismo determinar que es lo que está dentro del sistema y que está afuera.

2. Identificar la principal funcionalidad del sistema, mediante la definición de los casos de uso.

3. Determinar una posible solución para proponer una posible arquitectura.

4. Evaluar el costo, proponer un calendario de actividades e identificar los riesgos asociados al proyecto.

5. Decidir que proceso seguir y que herramientas utilizar.

4.3.2. Fase de elaboración

La fase de elaboración establece la arquitectura base del sistema y define las líneas que se seguirán para el desarrollo del proyecto, por lo que cuida enfocarse en lo más esencial de este. Se prefiere disminuir la probabilidad de desviar el proyecto o distraerse en elementos no esenciales del proyecto que retrasen su verdadero avance.

Lo que se está buscando es disminuir los riesgos asociados a: los requerimientos, la estimación de costos y a los cálculos relacionados con las actividades del proyecto. Por esto establecen los siguientes 4 objetivos:

1. *Obtener mayor detalle y entender claramente los requerimientos*. Esto se requiere para obtener una descripción de la mayor parte de los casos de uso del sistema, así como para obtener la descripción de los actores que participan en el sistema y en que caso de uso participan. En esta etapa se espera obtener el 80 % de los casos de uso asociados a sus actores. Además se genera un plan de actividades para el desarrollo del proyecto.

2. *Diseñar, implementar y validar la arquitectura del sistema para identificar los principales componentes con sus interfaces*. Esto no está limitado a dibujos o a un conjunto de diagramas, sino que es necesario una arquitectura ejecutable que pueda ser probada y validada con respecto al propósito básico del sistema. Podemos entender por arquitectura ejecutable, la implementación parcial del sistema, su construcción y la demostración de que el diseño arquitectónico soporta la función básica del sistema y sus propiedades en términos del desempeño de la arquitectura.

3. *Disminuir el impacto de los riesgo esenciales*. Considerando: personal, productos, procesos y el proyecto. En esta etapa se cuenta con la siguiente información:

   - Detalles de los requerimientos.
   - Un esqueleto de la estructura a implementar.
   - Un buen entendimiento de las personas que se han seleccionado para el sistema así como de las herramientas que se utilizarán.

   El resultado de esta etapa es un calendario con la estimación de costos del proyecto.

4. Refinamiento de los casos de desarrollo y establecimiento del ambiente de desarrollo.

4.3.3. Fase de Construcción

Esta es la tercera fase del ciclo de vida del RUP, la cual tiene un especial enfoque en los detalles del diseño, implementación y pruebas. Dado que la implementación tiene que ver con la funcionalidad del sistema será necesario revisar los detalles finos de los requerimientos.

Los objetivo de esta etapa son:

1. Generar un producto ejecutable para el usuario.

2. Minimizar el costo del desarrollo, buscando el trabajo en equipo, para optimizar recursos y evitar desacuerdos innecesarios.

4.3.4. Fase de Transición

Esta fase pone especial énfasis en revisar que el software que se elaboró esté dirigido a las necesidades del usuario. En esta etapa existe una diferencia radical entre el desarrollo de cascada y el RUP, ya que en esta etapa el RUP ya tiene un sistema estable, integrado y probado, mientras que en el modelo de cascada, en la fase de integración, aún no se tiene un avance significativo.

Sus principales objetivos son:

- Validar una sistema preliminar (versión beta) con las expectativas que fueron recopiladas.
- Entrenar a los usuarios y administradores, para que el sistema tenga cierta fiabilidad, por eso en este punto es necesario entrenar al personal de la empresa.
- Preparar el sistema para ser distribuido y vendido.
- Obtener el visto bueno del especialista, quien evaluara la versión para definir si se desarrolló el proyecto adecuadamente.
- Definir la evolución del proyecto.

4.4. Estructuras Estáticas

La estructura estática ayuda a definir: ¿quién hace que?, el ¿que hace?, ¿cómo? y ¿cuando lo hace?. Para esto propone una secuencia de actividades que se usan en cada fase de la estructura dinámica, de tal forma la estructura estática se convierte en los ciclos de las fases de la estructura dinámica. En la figura 4.5 se muestran los siguientes elementos de la estructura estática:

- Trabajadores (o roles que juega un usuario).

- Actividad.
- Artefactos de software.
- Flujos de trabajo.

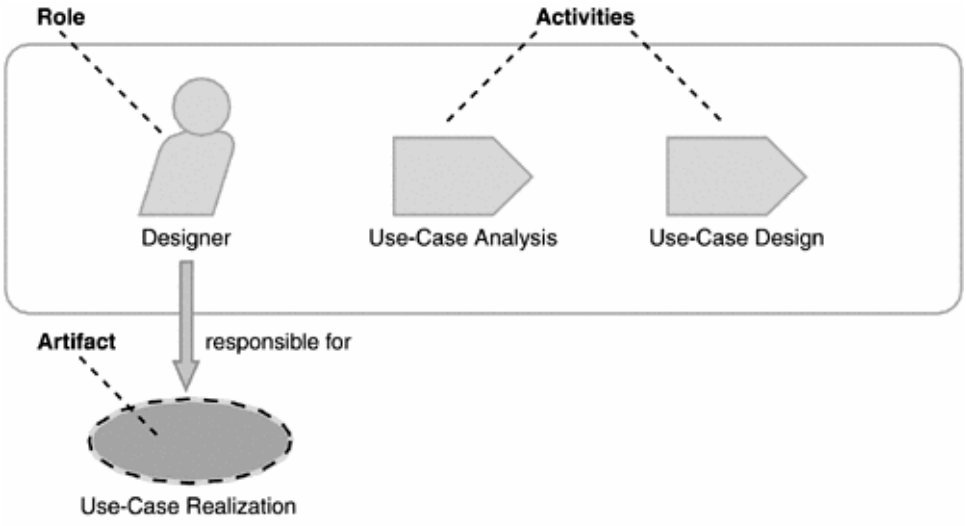


Figura 4.5: Elementos de la estructura estática

Cada uno de estos componentes ofrece cierta información para el proyecto.

- Trabajador: El quien.
- La actividades: El como.
- Los artefactos de software: El que.
- Los flujos de trabajo: El cuando.

A continuación explicaremos cada uno de estos.

4.4.1. Trabajador

Se puede entender por trabajador en el contexto del RUP, a un individuo o grupo de individuos con cierto comportamiento y responsabilidad. Los comportamientos se expresarán en terminos de las actividades desempeñadas por el trabajador. La responsabilidad de cada trabajador estará usualmente expresada en relación a ciertos artefactos de software que el trabajador

crea, modifica o controla. Se debe considerar que un trabajador puede desempeñar varios roles, o un rol especifico puede ser desempeñado por varios trabajadores. En la figura 4.6 se ilustra como se representa un trabajador.

Figura 4.6: Símbolo con el que se representa a un trabajador

4.4.2. Actividad

Una actividad es una unidad de trabajo que produce un resultado significativo en el contexto del proyecto. La actividad tiene un propósito claro, expresado generalmente en términos de crear o modificar artefactos de software, tales como un modelo, una clase, un código fuente o un plan. Cada actividad se asigna a un trabajador específico y los trabajadores tiene actividades las cuales definen su trabajo.

Dependiendo de la granularidad de una actividad, ésta puede tomar varias horas y hasta varios días y una actividad puede servir para planear los avances y estimar los progresos. Las actividades pueden ser representadas algunas veces por el mismo artefacto de software, especialmente de una iteración a otra, como un sistema que es refinado y expandido.

En términos de objetos, el trabajador es una actividad del artefacto de software, y la actividad es el desempeño del trabajador. En la figura 4.7podemos ver como se representa una actividad.

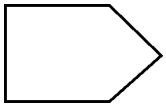


Figura 4.7: Símbolo con el que se representa a una actividad

4.4.3. Artefactos de Software

Es una pieza de información que es producida, modificada o usada para un proceso. Un artefacto de software es un producto tangible de un proyecto y es el principal producto de un proyecto. Los artefactos de software son usados como entradas para los trabajadores, para el desempeño de una actividad y son el resultado de la salida de una actividad.

Los artefactos de software pueden ser:

- El modelado de casos de uso o el modelado del diseño.
- Elementos de un modelo tales como: clases, casos de uso o un subsistema.
- Un documento. El caso de negocios o la arquitectura del modelo software.
- Código fuente.
- Programas Ejecutables.

4.4.4. Flujos de trabajo

La enumeración de todos los trabajadores, actividades y artefactos de software no necesariamente constituyen un proceso de software. Es necesaria una forma de describir una secuencia significativa de actividades que producen resultados valiosos y para mostrar la interacción entre los trabajadores. Un flujo de trabajo es una secuencia de actividades que produce un resultado de un valor observable. En términos de UML, un flujo de trabajo puede ser expresado como un diagrama de secuencia, colaboración ó actividades. Es importante destacar que no siempre es posible o práctico representar todas las dependencias entre actividades. A menudo dos actividades están estrechamente relacionadas, especialmente cuando incluyen al mismo trabajador.

El RUP usa los siguientes flujos de trabajo (ver figura 4.4):

- Modelo de Negocios.
- Requerimientos.
- Análisis y Diseño.
- Implementación.

- Pruebas.
- Desarrollo.
- Configuración y manejo de cambios.
- Manejo de proyecto
- Ambiente.

A continuación explicaremos cada uno de los flujos de trabajo del RUP.

4.4.4.1. Modelo de Negocios

Cuando se piensa en crear cierto software, se parte de un proceso existente o propuesto. Sin embargo es de vital importancia entender los procesos, las reglas de operación, el lenguaje de los dueños del proceso y el rol que juegan los usuarios que participan en dicho proceso. El principal objetivo del modelo de negocios es:

- Entender la estructura y la dinámica de la organización en la cual el sistema trabajará.
- Entender los problemas en la organización e identificar las mejoras potenciales.
- Asegurar que el cliente, el usuario final y el desarrollador tienen claro el objetivo de la organización.
- Manejar los requerimientos necesarios para soportar los objetivos de la organización.

El modelo de negocios describirá como obtener la visión de los objetivos de la organización y basado en esta visión, definirá los procesos, los roles, y las responsabilidades de la organización. Este modelo está comprometido con los casos de uso y con el modelo del objeto de negocios.

Para facilitar la comprensión del modelo de negocios y para ubicarlo en el contexto del problema a resolver, se visualiza el modelo de negocios y se proporciona un lenguaje gráfico que ilustra la interacción entre los participantes de este modelo, los cuales son: Los usuario del modelo, su rol que desempeñan, los procesos que integran el modelo y los objetos o artículos que se producen por la organización.

Dado que el modelo de negocios puede tener diferentes alcances dependiendo del contexto y las necesidades, listaremos seis posibles escenarios.

- *Escenario 1: Cuadro sinóptico*. Se puede hacer un simple cuadro sinóptico de la organización y de sus procesos, de los requerimientos ya recopilados para la aplicación que se quiere construir.

- *Escenario 2: Dominio del modelo*. Si se construye una aplicación con el propósito de manejar y presentar información, se puede construir un modelo de negocios por niveles, sin considerar el flujo de trabajo. A esto es a lo que se le llama dominio del modelo y éste es típicamente parte de un proyecto de Ingeniería de Software.

- *Escenario 3: Un modelo de negocio, varios sistemas*. Si se construye un sistema grande ó una familia de aplicaciones, se puede tener un modelo de negocios que se desarrolle con varios proyectos de ingeniería de software. Este tipo de modelos ayuda a encontrar los requerimientos funcionales, además de proporcionar las entradas para la creación de la arquitectura de una familia de aplicaciones.

- *Escenario 4: Modelo de negocios genérico*. Si se construye una aplicación que será usada por varias organizaciones, con un modelo de negocios similar, se debe evitar requerimientos especializados o complejos para el sistema. Sin embargo, si no existe la opción de omitir estos requerimientos, es importante destacar las diferencias entre los requerimientos de las diferentes organizaciones, con el fin de buscar prioridad en la funcionalidad de la aplicación.

- *Escenario 5: Nuevo proceso*. Si una organización ha decidido una nueva línea de negocios compleja, esta deberá construir el sistema de información que la soporta. Esto no es solo con el objetivo de obtener los requerimientos del modelo de negocios, sino el de determinar que tan factible es la nueva línea de negocios. A menudo esta nueva línea de negocios se trata como un proyecto con su propio modelo de negocios.

- *Escenario 6: Modernizar*. Si la organización ha decidido modernizar sus procesos, se tendrá que pensar en un proceso de reingeniería para obtener el modelo de negocios e

introducir los cambios que sean necesarios.

En la figura 4.8 se muestra un diagrama del proceso utilizado para la creación del modelado del negocio. Se pueden tomar varios caminos para este proceso dependiendo del objetivo de su modelado de negocios, así como de su posición en el ciclo de vida del desarrollo. En la primera

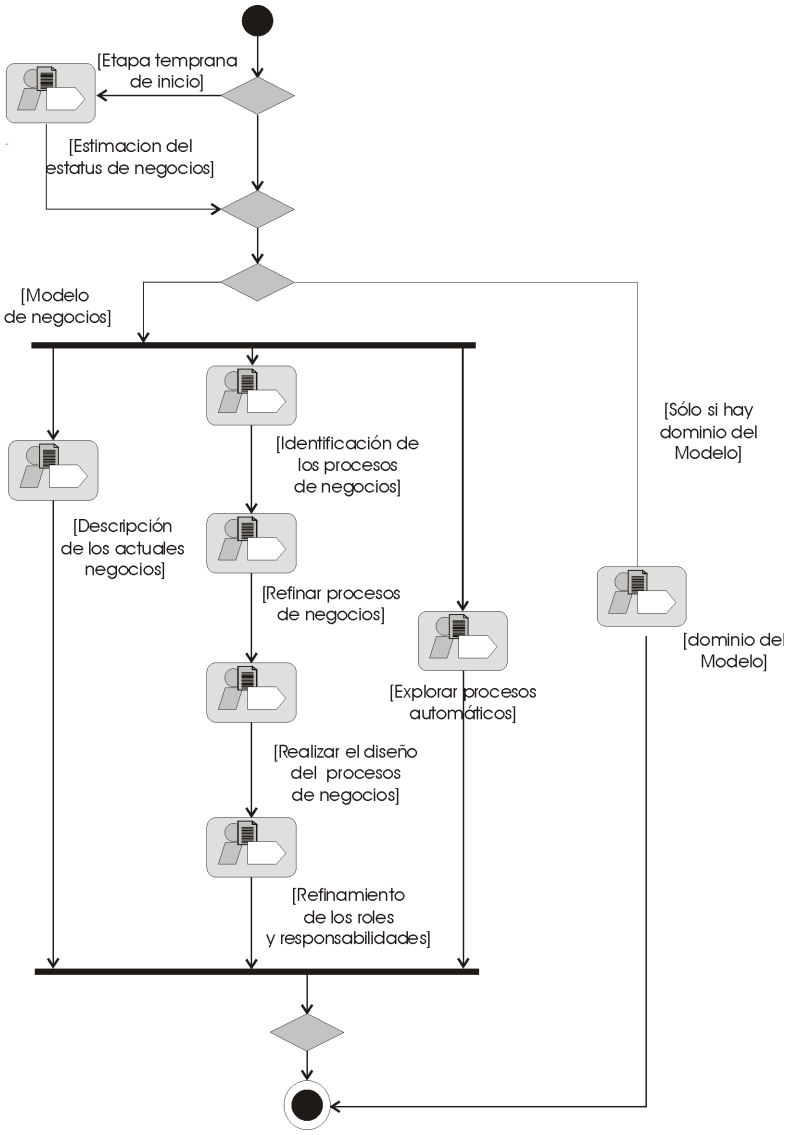


Figura 4.8: Flujo de trabajo para el modelo de negocios

iteración se evaluará el estado de la organización. Los artefactos primarios producidos aquí son

la evaluación de los objetivos de la organización y la visión de negocio.

Basados en los resultados de las evaluaciones, se decidirá cuál de los escenarios del modelado de negocios se seguirá y entonces se podrá tomar decisiones sobre como seguir en la iteración y también sobre como trabajar en iteraciones subsecuentes.

Si se determina que no se necesitan modelos a escala de negocio, (escenario 2), se seguirá el dominio que modela el camino de este proceso laboral.

Si se determina que los procesos de negocios no requieren cambios, entonces se deberán diagramar los procesos existentes y manejar los requerimientos del sistema, enfocándose directamente en los objetivos actuales de la organización y siguiendo cada una de las tareas del modelo de negocios.

Si el modelado del negocio se lleva a cabo con la intención de mejorarlo o de usar reingeniería de un negocio existente (Escenarios 3, 4, y 6), sería necesario modelar tanto el negocio actual como el nuevo negocio.

Si el modelado del negocio se lleva a cabo con la intención de desarrollar un nuevo negocio más o menos desde el principio (escenario 5), sería necesario prever el nuevo negocio y construir los modelos del nuevo negocio, pero faltaría la “descripción del negocio actual”.

4.4.4.2. Requerimientos

Los requerimientos de software se definen como los servicios que el usuario desea que proporcione cierto sistema y las restricciones sobre las que el software debe operar.

Tradicionalmente, los requerimientos son vistos como una especificación textual detallada, expresada en la forma: “El sistema deberá...”.

El propósito de la obtención de los requerimientos en el proceso de RUP es el siguiente:

- Establecer y mantener acuerdos con los clientes y otros expertos del modelo de negocios (stakeholders) con respecto a: ¿Qué es lo que debe hacer el sistema? y ¿por qué?
- Proveer a los desarrolladores del sistema de un mejor entendimiento de lo que se desea que haga el sistema.

- Definir las fronteras del sistema (delimitarlo).
- Proveer las bases para la planeación del contenido técnico de las iteraciones.
- Proveer las bases para la estimación de costos y el tiempo de desarrollo del sistema.
- Definir una interfaz de usuario para el sistema, enfocándose en las necesidades y objetivos de los usuarios.

El proceso para manejo y obtención de requerimientos se muestra en la figura 4.9.

Descripción del flujo de trabajo El flujo de actividades descritos en la figura 4.9 permite la obtención de los requerimientos, por lo que explicaremos en que consiste cada una de estas actividades:

1. *Analizar el Problema*: Se obtiene un acuerdo con respecto al problema a resolver, identificando a los Stakeholders, las fronteras y las limitaciones del sistema.

2. *Entender las necesidades del stakeholder*: Se utilizan varias técnicas de licitación para, reunir los pedidos del Stakeholder y obtener una comprensión clara de las verdaderas necesidades de los usuarios y de los Stakeholders del sistema.

3. *Definir el sistema*: Se basa en las entradas de los stakeholders para establecer un conjunto de características del sistema. También se determinan los criterios que se utilizarán para priorizar las entregas, definiendo lo que se entregará identificando los casos de uso principales para la entrega.

4. *Manejo del alcance del sistema*: Se reúne la información importante de los stakeholders para priorizar y delimitar el conjunto de requerimientos acordados para entregarlo dentro del tiempo y costo presupuestado.

5. *Refinar la definición del sistema*: Se detallan los requisitos del sistema de software para acordar con el cliente la funcionalidad del sistema, y capturar otros requisitos importantes.

6. *Manejo del cambio de requerimientos*: Para controlar los cambios de requerimientos se ocupa una tabla central donde se documentan los cambios, la cual tiene como objetivo

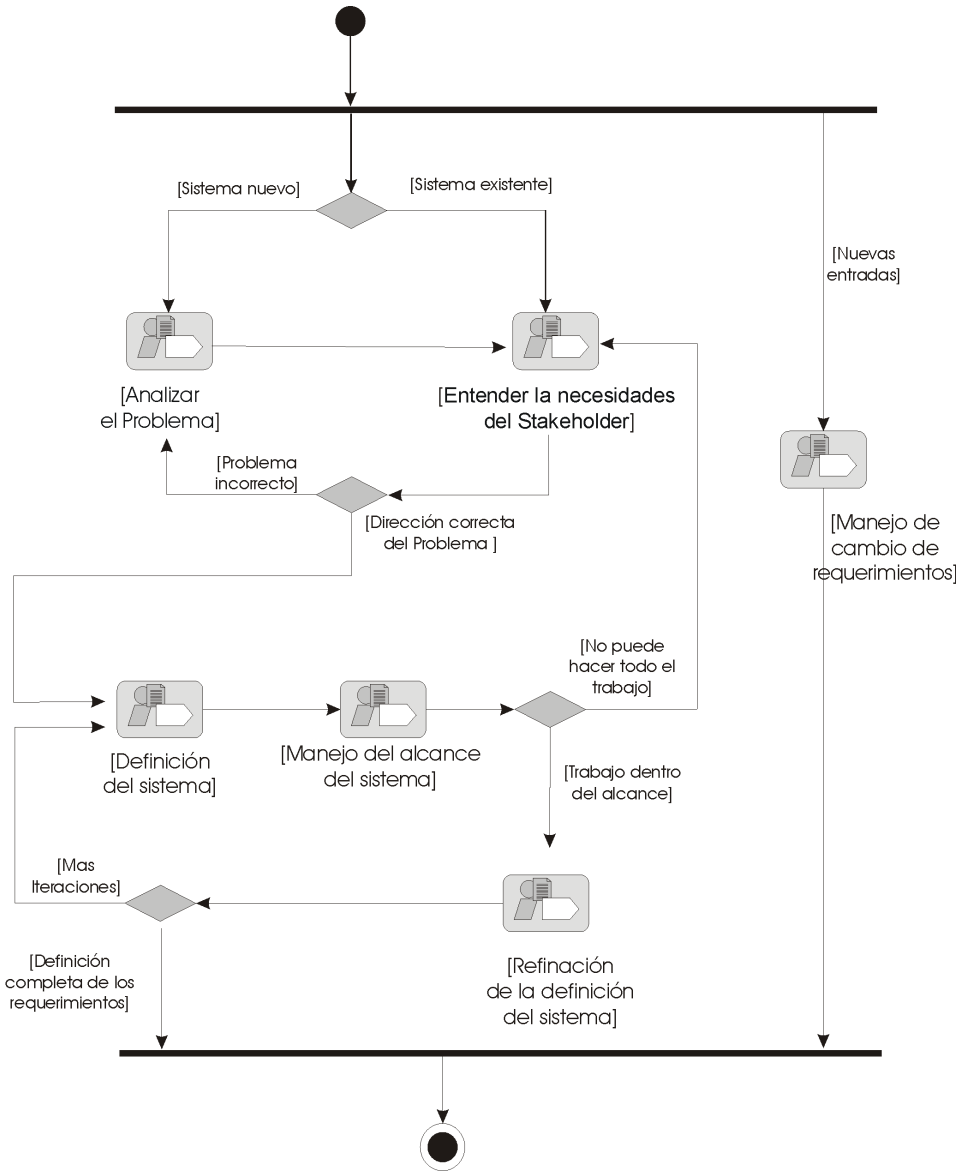


Figura 4.9: Flujo de trabajo para manejo y obtención de requerimientos

verificar el impacto de cambiar un requerimiento. Los cambios a los requerimientos deben realizarse de común acuerdo con el cliente.

4.4.4.3. Análisis y Diseño

El propósito del análisis y diseño es traducir los requerimientos en especificaciones que describan cómo se implementa el sistema. Para realizar esta traducción se deben entender los requerimientos y transformarlos a un diseño, seleccionando las mejores estrategias para su implementación. Dentro del proyecto se debe establecer una arquitectura robusta para poder diseñar un sistema que facilite su comprensión, construcción y evolución. Después, se debe ajustar el diseño para acoplarlo con el entorno de implementación, diseñándolo para que sea ejecutable, estable, escalable y que permita poner a prueba su funcionamiento.

La figura 4.10 describe las siguientes actividades para realizar el análisis y el diseño.

1. *Definir la arquitectura*. Esta actividad es la base para un buen diseño en la creación de un sistema, por tal razón hay que considerar:
   - Crear un bosquejo inicial de la arquitectura del sistema, definiendo:
     - Un conjunto inicial de elementos arquitectónicamente significativos para ser utilizado como la base para el análisis.
     - Un conjunto inicial de mecanismos de análisis.
     - Un acuerdo inicial de la organización del sistema.
     - La realización de caso de uso para ser dirigidos en la iteración actual.
   - Identificar el análisis de las clases de la arquitectura de los casos de uso.
   - Actualizar los casos de uso para la definición de la arquitectura
   - Cuidar la consistencia e integridad de la arquitectura

2. *El Análisis de comportamiento*. Se compone del análisis de actividades de los casos de uso, realizados por el diseñador, identificando los elementos de diseño (realizado por el arquitecto), y revisando el diseño (realizado por el supervisor del diseño). El propósito es transformar las descripciones proporcionadas por los casos de uso en un conjunto de elementos, los cuales son la base del diseño.

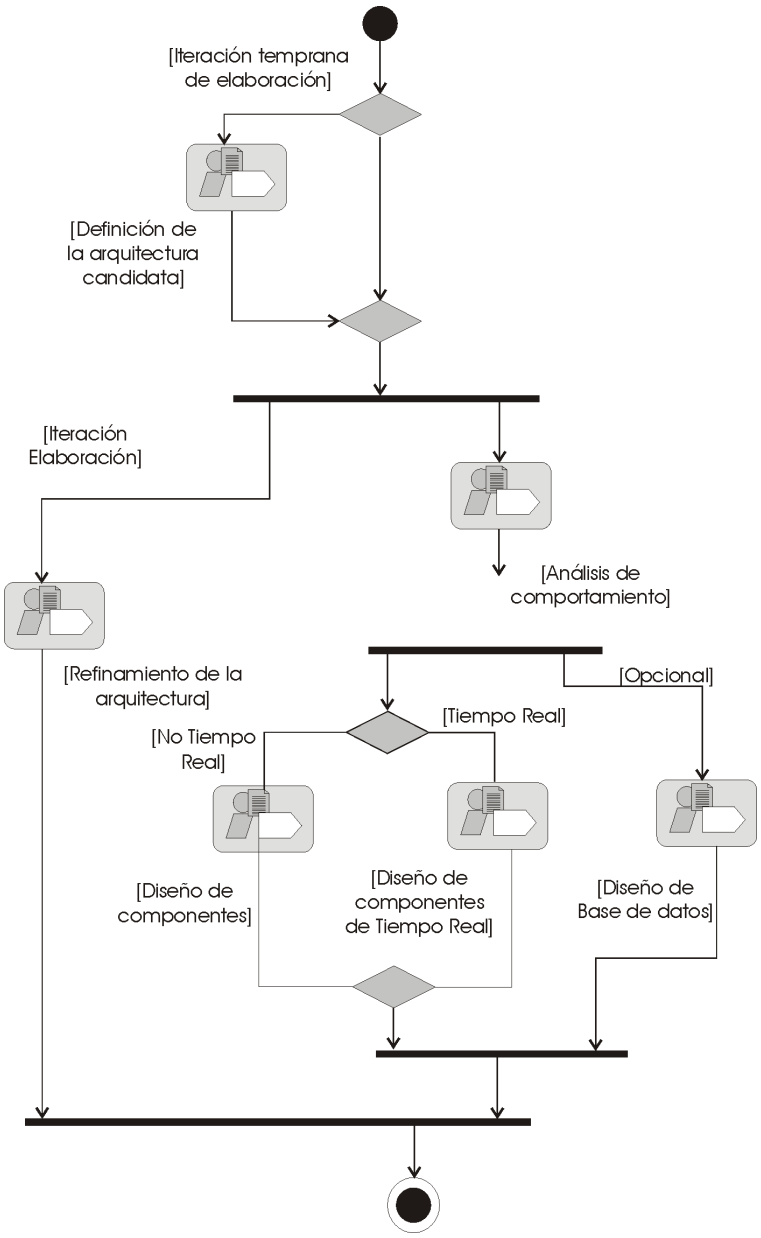


Figura 4.10: Flujo de trabajo para manejo del Análisis y Diseño

3. *El Diseño de Componentes*. Se basa en los casos de uso, en los paquetes o en los subsistemas. Se debe especificar el diseño de los componentes y sus características, la relación de sus clases y las interfaces de los subsistemas. Su principal objetivo es:

- Diseñar como se implementará cada elemento de los componentes, indicando los detalles de la implementación así como la conducta de los elementos.
- El diseño se basa en los casos de uso, por lo que es necesario refinarlos y actualizarlos.
- Revisar el diseño conforme éste evoluciona.

4. *Diseño de la base de datos*. Esta actividad es opcional, dado que se ocupa cuando el sistema maneja información que debe ser almacenada. Su propósito es:

   - Identificar las clases persistentes en el diseño.
   - Diseñar estructuras de base de datos apropiadas para almacenar las clases.
   - Definir los mecanismos y estrategias para almacenar y recuperar los datos persistentes.

4.4.4.4. Implementación

La etapa de trabajo de implementación tiene 4 propósitos:

- Definir la organización del código en términos de subsistemas organizados en capas.
- Implementar clases y objetos en términos de componentes.
- Evaluar los componentes desarrollados como unidades.
- Integrar en un sistema ejecutable los resultados producidos por los programadores individuales o equipos.

En esta etapa se deben evaluar las unidades o componentes individuales (el programador es responsable de evaluar cada unidad).

El flujo de actividades que se observa en la figura 4.11 muestra el proceso de implementación, el cual describiremos a continuación.

1. Planear qué subsistemas se van a integrar y en qué orden

2. Para cada subsistema, el programador responsable planea la integración del subsistema.

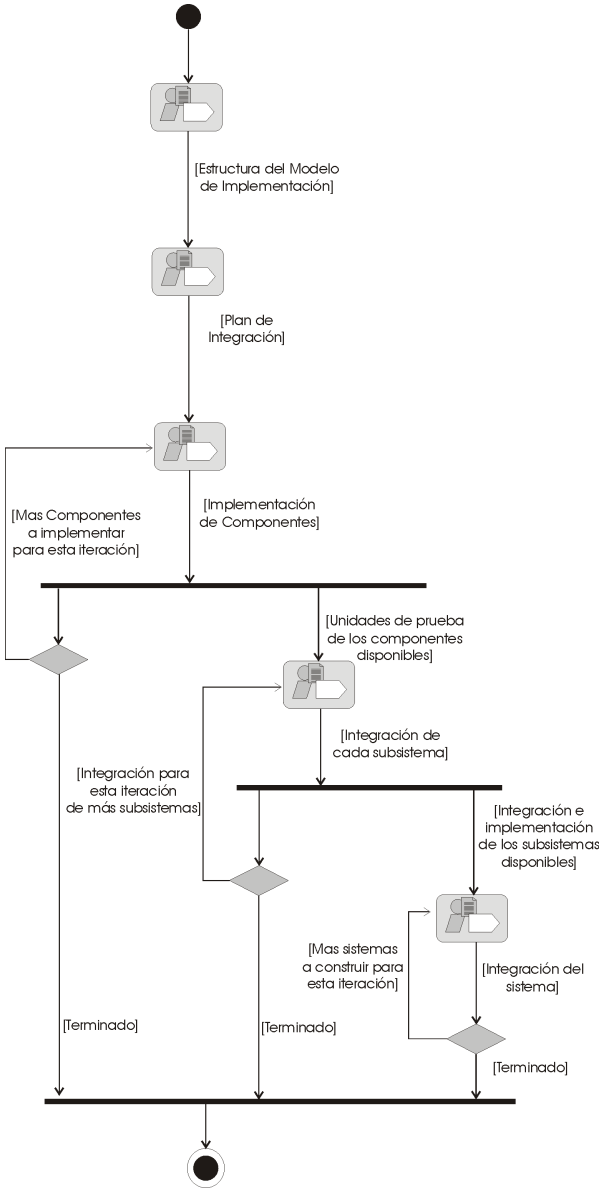


Figura 4.11: Flujo de trabajo para manejo la Implementación

3. Los programadores implementan las clases y objetos en el modelo de diseño. Se compilan y generan las ligas y se ejecutan. Si los programadores descubren defectos en el diseño, se re-elabora el diseño.

4. Los programadores arreglan defectos de código y hacen pruebas de unidades para verificar

los cambios. Así mismo el código es revisado para asegurar la calidad.

5. Si varios programadores están trabajando en la implementación del mismo subsistema, uno de los programadores se encarga de integrar los cambios hechos en una nueva versión del subsistema.

6. Se hacen nuevas construcciones a partir de los resultados de esta integración, las cuales son evaluadas por un evaluador de integración.

7. La última versión del subsistema es liberada para la integración del sistema.

4.4.4.5. Pruebas

El propósito de hacer pruebas es poder estimar la calidad del producto. Esto no sólo se aplica al producto final, si no que comienza desde el inicio del proyecto con la valoración de la arquitectura y continua durante todo el desarrollo del proyecto hasta llegar a la valoración del producto final entregado a los clientes.

Los objetivos de la fase de pruebas son:

- Verificar las interacciones de los componentes.
- Verificar la correcta integración de los componentes.
- Verificar que todos los requerimientos hallan sido implementados correctamente
- Identificar y asegurarse de que todos los errores sean documentados antes de que el software sea desarrollado.

El flujo de actividades que se observa en la figura 4.12 describe el proceso de de pruebas.

1. *Planear la Prueba*. El propósito de hacer una planeación para las pruebas es identificar y describir las pruebas que serán implementadas y ejecutadas. Esto se logra creando un plan de pruebas que contenga los requerimientos de las pruebas y las estrategias de las mismas. Un plan de pruebas puede ser desarrollado describiendo todos los tipos de pruebas que se implementarán y ejecutarán, haciendo un plan de pruebas por tipo de prueba que se podría realizar.

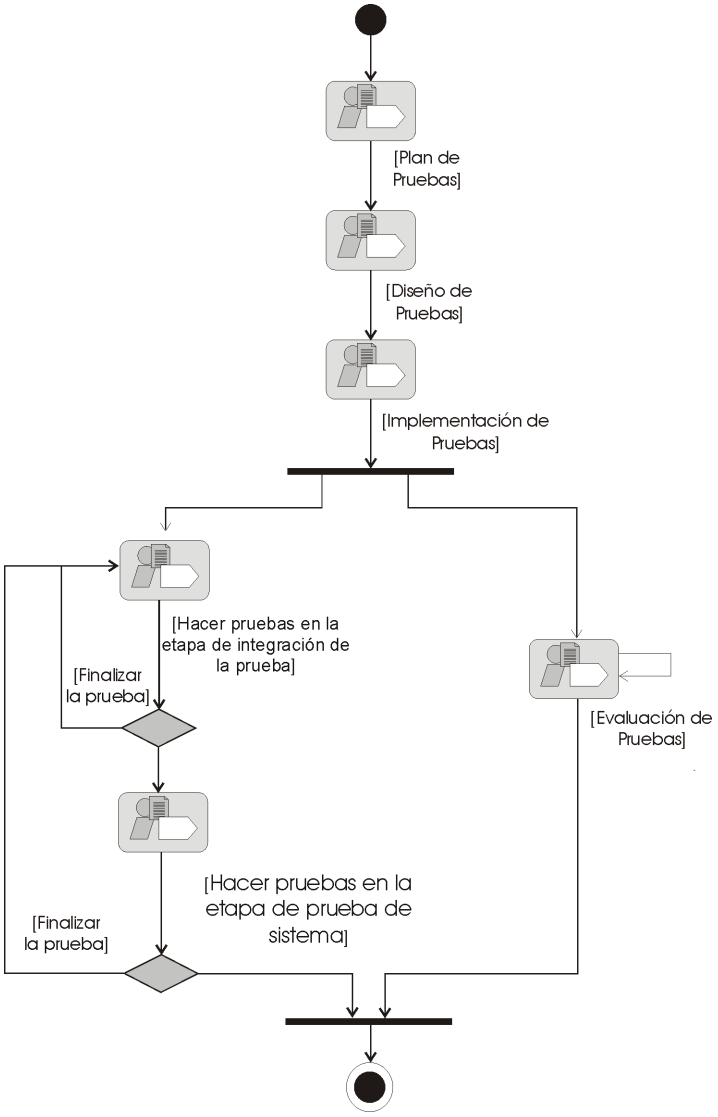


Figura 4.12: Flujo de trabajo para manejo de pruebas

2. *Diseño de Pruebas*. El propósito de diseñar las pruebas es identificar, describir y generar los modelos de pruebas y sus artefactos reportados. El diseño de las pruebas se hace para asegurar que el software cumpla con sus requerimientos y que sea estructurado y organizado correctamente. En la actividad de diseño, el diseñador de las pruebas analiza el objetivo de cada prueba y desarrolla el modelo de pruebas.

El diseño de las pruebas transforma los casos de uso en casos de prueba de aprobación, y las actividades de los casos de uso, en casos de prueba de: unidad, integración y sistema.

3. *Implementar las Pruebas*. Se implementan los procedimientos de las pruebas que fueron definidos en la sección de diseño de pruebas. La creación de las pruebas es hecha en el contexto de una herramienta de automatización o en un ambiente de programación. El artefacto generado es una versión del procedimiento de pruebas que puede ser leído en una computadora o referido como un manuscrito de pruebas.

4. *Hacer pruebas en la etapa de integración*. El propósito de la etapa de pruebas en la integración es para garantizar que durante el ensamblado, los componentes del sistema colaboren como se especifica, así como asegurar que el incremento de cada módulo tenga un comportamiento correcto. El integrador del sistema compila y une el sistema en incrementos. Por cada incremento, la funcionalidad que ha sido agregada es probada. Así mismo se realizan pruebas de regresión, con los resultados de las pruebas.

5. *Hacer pruebas del sistema*. El propósito de la etapa de pruebas del sistema es garantizar que el sistema completo (según los objetivos de la iteración) funcionó como se pretende. Durante una iteración, las pruebas del sistema se realizan varias veces hasta que el sistema completo (como es definido por el objetivo de la iteración) haya funcionado como se especificó y logré el éxito de las pruebas o el criterio esté completo.

6. *Evaluar la prueba*. El propósito de evaluar una prueba es entregar y asegurar las medidas cuantificables para determinar la calidad del objetivo de prueba y la calidad del proceso de la prueba. Esto se logra revisando y evaluando los resultados de la prueba, identificando y agregando las solicitudes de cambios.

4.4.4.6. Configuración y Control de Cambios

El propósito de la configuración y el control de los cambios, es dar seguimiento y mantener la integridad de la evolución del proyecto. Durante el desarrollo del ciclo de vida, muchos

artefactos valiosos son creados. La labor del desarrollo de estos componentes es intensa, y representa una inversión significativa. La evolución y actualización de los artefactos y componentes se dará constantemente en el desarrollo iterativo. Aunque usualmente un trabajador es el responsable de un artefacto, no podemos confiar en la memoria de este trabajador.

Los miembros del equipo del proyecto deben ser capaces de identificar y localizar los artefactos, de seleccionar la versión apropiada de un artefacto, de observar su historial y entender su estado actual y las razones por las cuales ha cambiado el proyecto y de comprobar quien es el responsable de los cambios. Al mismo tiempo el equipo del proyecto debe dar seguimiento a la evolución del producto, capturar y manejar los pedidos de cambios sin importar de donde vienen, e implementar los cambios en forma consistente a través de los distintos artefactos.

El flujo de actividades que se observa en la figura 4.13 describe el proceso de la configuración y control de cambios.

1. *Plan de configuración del proyecto y el control de cambios*. El plan del manejo de la configuración describe todas las actividades relacionadas con el Manejador de Cambios (CM) que necesitan repetirse durante el curso del proyecto. Este describe los procedimientos y políticas aplicadas para identificar, proteger y reportar todos los artefactos que fueron generados durante el ciclo de vida del proyecto. Las convenciones en el nombramiento del proyecto son importantes porque facilitan la comunicación y permiten la fácil identificación de la configuración de los artículos para su actualización o reuso. Los artefactos del proyecto son activos tangibles que necesitan ser protegidos. La protección es lograda mediante el refuerzo de la archivación, los respaldos y los privilegios del control de acceso.

   El plan CM también describe el proceso de control de cambios del proyecto. El propósito de este proceso es asegurar que todos los cambios planeados ó realizados a un artefacto del proyecto son realizados informando a los a todos los involucrados en el proyecto, a los administradores y compañeros de trabajo que necesitan saber acerca de la naturaleza exacta del cambio.

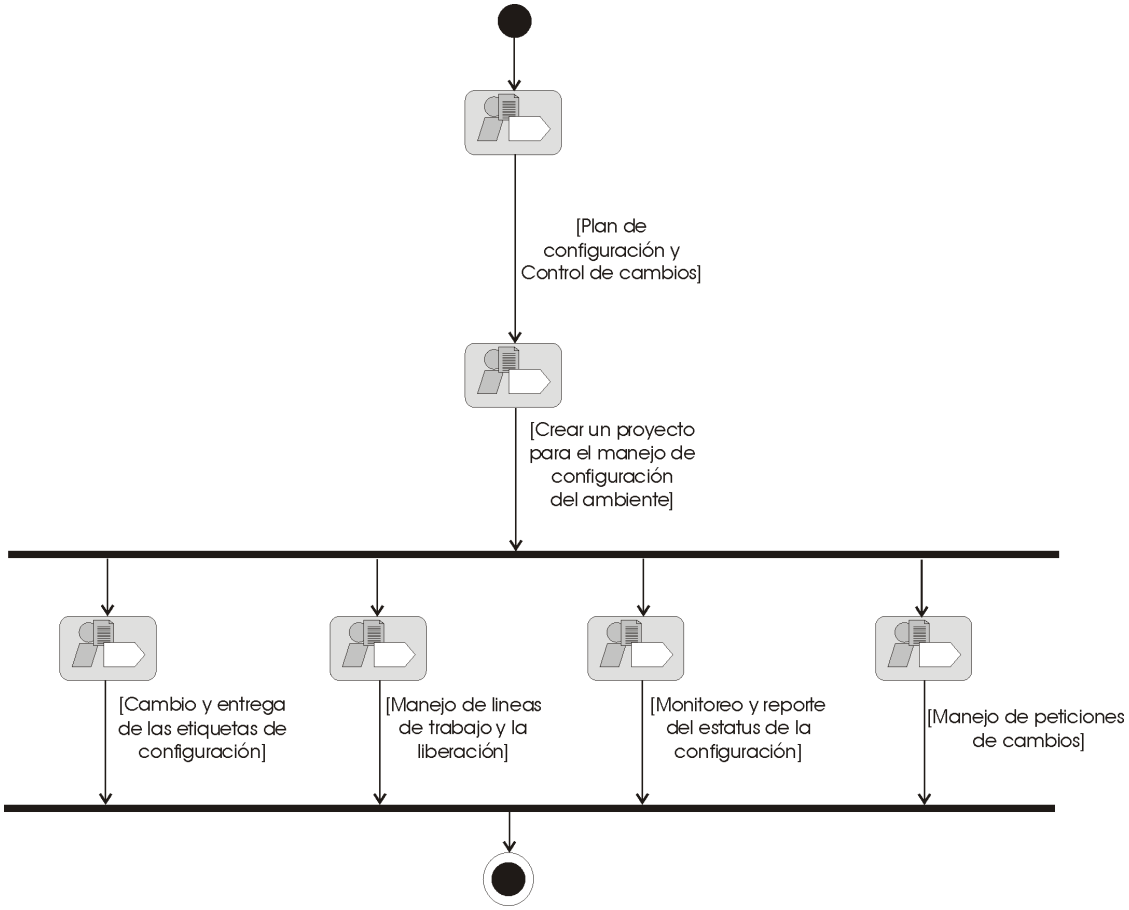


Figura 4.13: Flujo de trabajo para la Configuración y Control de Cambios

2. *Crear un proyecto para el manejo de configuración del ambiente*. Esto podría facilitar el desarrollo, ya que todos los trabajadores pueden obtener los artefactos de software de alguna parte del sistema, si así lo necesitan. Sin embargo esto implica que conforme algún desarrollador está trabajando este también realice sus artefactos de software.

   Es importante crear un espacio de trabajo donde los desarrolladores tengan a su alcance artefactos de software, donde ellos los pueden almacenar, donde se prueben los desarrollos y se integren, para su mantenimiento o su reutilización.

3. *Cambio y entrega de los artículos de la configuración*. Se refiere a la forma en la que cual-

quier trabajador puede crear un espacio de trabajo. Dentro de este espacio, el trabajador puede tener acceso a artefactos del proyecto, hacer cambios a esos artefactos y entregar los cambios para que se incluyan en el producto total. La liberación de cambios es realizada en un ambiente de trabajo de integración que puede incluir los envíos de múltiples trabajadores. La idea es integrar las contribuciones individuales y hacerlas visibles a los desarrolladores para su consulta. El integrador del sistema, desde el espacio de integración, construye el producto, define las líneas de trabajo y las pone a disposición del resto del equipo de desarrollo.

4. *Manejo de las líneas de trabajo y liberación*. Una línea de trabajo es una descripción de todas las versiones de los artefactos que constituyen al producto en un tiempo dado. Las líneas de trabajo son creadas al final de las iteraciones y en los puntos de entrega del proyecto. Un producto debe ser delimitado cada vez que se entrega al cliente. Cuando un cliente tiene un problema con una entrega (versión) el equipo de desarrollo o mantenimiento puede regresar a una línea trabajo o delimitación para recrear y arreglar el defecto particular para su liberación.

5. *Monitoreo y reporte del estatus de configuración*. El ambiente CM provee el contexto para todo el trabajo de desarrollo de software. Todos los artefactos necesitan ser puestos bajo un control de configuración y el administrador del CM quien es responsable de crear espacios de trabajo donde los desarrolladores e integradores realicen su trabajo.

   El administrador del CM tiene que asegurarse de que están seguros y de que se tienen todos los artefactos necesarios. El administrador del CM también es responsable de reportar el estado de la configuración. El reporte de la configuración puede proporcionar suficiente información al administrador del proyecto para el seguimiento total del progreso del proyecto y las tendencias. Esto ayudar a determinar donde se encuentran los problemas y proporciona información para determinar el impacto en los costos y calendarización del proyecto.

6. *Manejo de las peticiones de cambio*. El propósito de un proceso estándar de documentación de control de cambios es asegurar que los cambios en un proyecto son realizados de manera consistente y los stakeholder apropiados estén informados sobre, el estado del producto, los cambios a este y el impacto en el costo y calendarización de estos cambios.

4.4.4.7. Ambiente

El propósito del ambiente es dar soporte a la organización en el desarrollo con los procesos y herramientas. El soporte incluye:

- Herramientas de selección y adquisición.
- Herramientas de configuración.
- Proceso de configuración.
- Proceso de implementación.
- Servicios técnicos para mantener el proceso de la: información tecnológica (IT), infraestructura, informe de la administración y respaldo.

El flujo de actividades que se observa en la figura 4.14 describe el proceso para la generación del ambiente.

1. *Preparar el ambiente del flujo de trabajo*. El propósito de prepara el ambiente del flujo de trabajo para un proyecto es para:
   - Evaluar el actual ambiente de la organización.
   - Evaluar las actuales herramientas de soporte.
   - Desarrolla un primer proyecto del caso de desarrollo.
   - Producir una lista de herramientas candidato, para ser usadas en el desarrollo.
   - Produce una lista plantillas de especificaciones del proyecto candidatas para los artefactos clave.

2. *Preparar el ambiente para una iteración*. El propósito de preparar un ambiente para una interacción incluye lo siguiente:

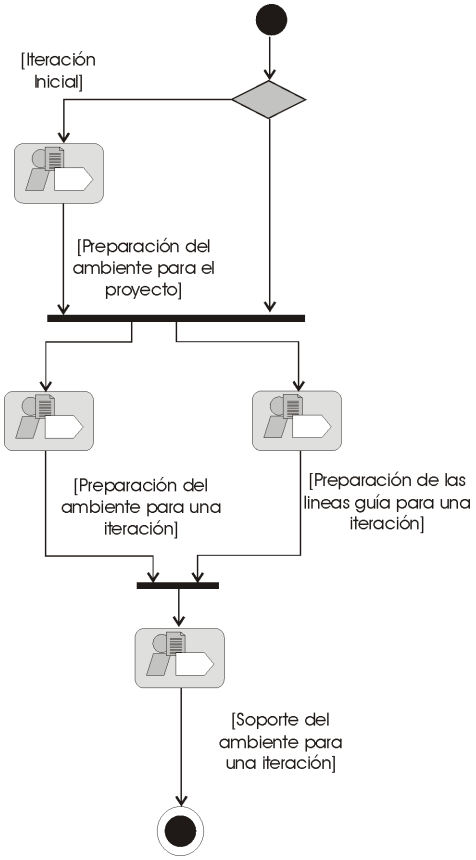


Figura 4.14: Flujo de trabajo para el Ambiente

- Completar el caso de desarrollo para estar listo para la iteración.
- Preparar y si es necesario personalizar herramientas para usar dentro de una iteración.
- Verificar que las herramientas se han configurado y se han instalado correctamente.
- Producir un juego de plantillas de especificaciones del proyecto para usar dentro de la iteración.
- Entrenar a las personas para usar y entender las herramientas y el caso de desarrollo.

3. *Preparar pautas para una iteración.*

4.4.4.8. Desarrollo

No basta con que un programa compile libre de defectos, también es importante también considerar a los usuarios antes de pensar que se ha logrado el objetivo. Por esto, es importante considerar las siguientes actividades:

- Probar el software en su ambiente operacional final (prueba beta).
- Empacar el software para entrega.
- Distribución del software.
- Instalación del software.
- Capacitación a los usuarios finales.
- Fuerza de la venta.
- Migración a software existentes y modificación de las bases de datos

Las forma en que estas actividades son llevadas a cabo en distintas industrias del software, depende del tamaño del proyecto, del modo de entrega y del contexto del negocio.

El flujo de actividades que se observa en la figura 4.15 describe el proceso para desarrollo.

1. *Plan del desarrollador*. El planteamiento del desarrollador, debe considerar no solamente el ¿cómo? y ¿cuando? la lista de requisitos será desarrollada, sino también debe asegurarse de que el usuario final tenga toda la información necesaria para tomar la entrega del nuevo software. Para asegurar una transición adecuada, los planes del desarrollador deben incluir el programa de la prueba beta iterativa de la versión actual en construcción.

   Una conclusión acertada a un proyecto de software, puede verse afectada seriamente por factores fuera del alcance del desarrollo del software, como puede ser la infraestructura del hardware, usuarios incapacitados para el acceso al nuevo sistema, etc.

2. *Material de apoyo para el desarrollo*. El material de apoyo cubre un amplio grado de información que será requerida por el usuario final para instalar, operar y mantener el sistema desarrollado. Esto también incluye material de entrenamiento para todas las variadas posiciones que serán requeridas en el uso efectivo del nuevo sistema.

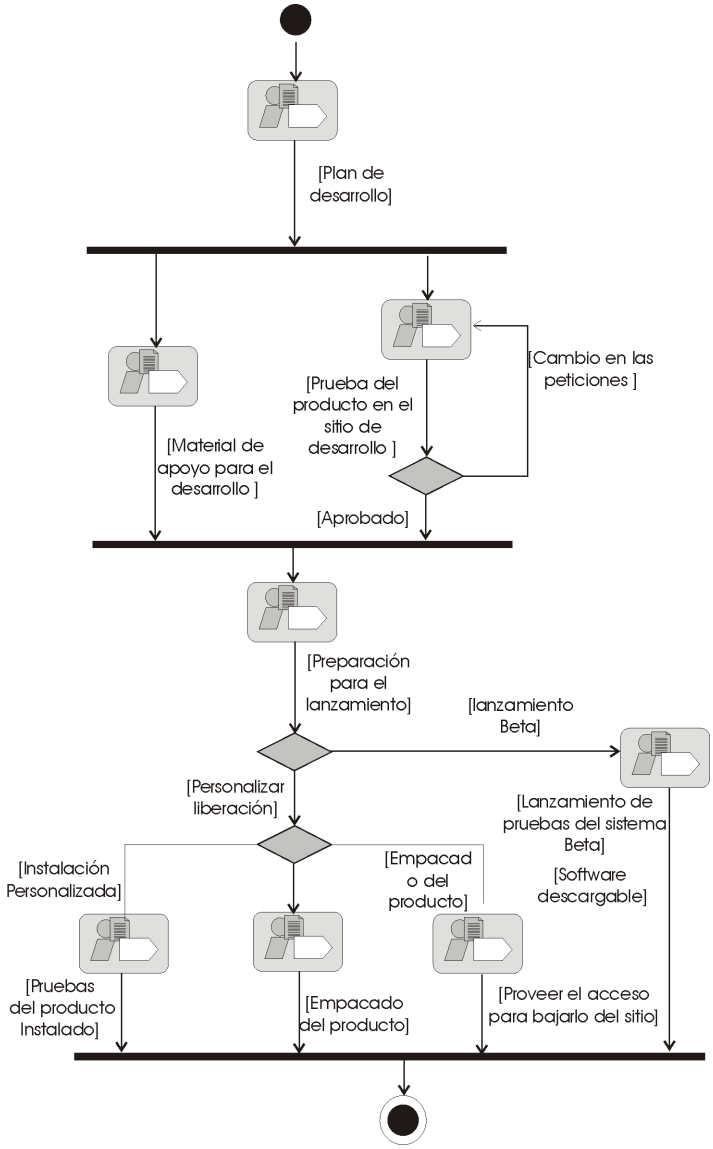


Figura 4.15: Flujo de trabajo para el Desarrollo

3. *Prueba del producto en el sitio de desarrollo*. Probar en el sitio de desarrollo, ayuda a determinar, si el producto es suficientemente maduro para el lanzamiento, así como para la entrega final o para la distribución a los probadores beta.

   La prueba beta de un producto empacado es hecha por un amplio rango de usuarios finales, a fin de obtener su retroalimentación. En el caso de un cliente de sistema de instalación,

una prueba beta puede ser un piloto de instalación en la plataforma.

4. *Creadores de lanzamiento*. El propósito es asegurarse de que el producto está preparado para la entrega al cliente. Un lanzamiento consiste en cuidar todo lo necesario para que los usuarios finales instalen y ejecuten el software.

5. *La prueba beta*. El lanzamiento de la prueba beta requiere, que el software entregado sea instalado por el usuario final, que proporcionara la retroalimentación en su funcionamiento y utilidad.

6. *Prueba del producto en el sitio de la instalación*. Basándose en las pruebas de la iteración y los contratos precedentes del cliente, las pruebas finales del sitio no deben sorprender, deben ser solamente una formalidad para que el cliente “acepte”el sistema.

7. *Empacado del producto*. Este conjunto de actividades opcionales describe qué necesidades de empaquetamiento, para producir “productos del software”. En este caso, se contempla la producción en masa y después se empaqueta el producto para el lanzamiento en cajas, con los materiales necesarios para ser enviados al los clientes.

8. *Proveer el acceso al sitio*. Los desafíos actuales marcan que la mayoría de productos de software deben poderse distribuir a través de la red.

Capítulo 5

Lenguaje de Modelado Unificado

UML son las siglas de Lenguaje de Modelado Unificado (Unified Modeling Language UML). El UML es un lenguaje para especificar, visualizar, construir y documentar los artefactos de software. El UML representa una colección de buenas practicas de ingeniería que han provisto éxito en el modelado de largos y complejos sistemas.

El UML es un lenguaje de modelado orientado a objetos. Se debe recalcar que el UML no es una metodología, aunque proporciona técnicas que pueden ser usadas en conjunto o parcialmente en metodologías, fundamentalmente aquellas destinadas al desarrollo orientados a objetos.

Al ser UML un lenguaje, está compuesto de una sintaxis y una semántica.

El vocabulario de UML se compone de:

- Elementos: Son abstracciones de algún componente o aspecto del sistema modelado.
- Relaciones: Ligan elementos entre sí.
- Diagramas: Agrupan elementos y relaciones.

5.1. Elementos

Hay cuatro tipos de elementos:

1. Los Elementos estructurales: Representan partes materiales o conceptuales del sistema,

los elementos estructurales principales de UML son:

- La clase.
- La interfaz.
- La colaboración.
- El caso de uso.
- Los componentes.
- Los nodos

2. Los elementos de comportamiento: Son las partes dinámicas de los modelos UML. Los elementos de comportamiento básico son:

   - Las interacciones, que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular, para alcanzar un propósito específico.
   - La máquina de estados, que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos.

3. Los elementos de agrupación: Son los paquetes, y sirven para organizar otros elementos de los modelos UML. Un paquete es un mecanismo conceptual de propósito general para organizar elementos en grupos. Por ejemplo, se pueden incluir en un paquete varias clases y luego asignar ese paquete a un componente, con esto lo que se esta haciendo es asignar todas las clases al componente a través de un paquete.

4. De anotación: Los elementos de anotación son partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clarificar o hacer observaciones sobre cualquier elemento de un modelo.

5.2. Relaciones

Los tipos fundamentales de relaciones son los siguientes:

- Generalización: Conecta a elementos vinculados por una relación padre/hijo (generalización/ especialización). Gráficamente viene dada por una línea acabada en punta de flecha vacía que va del elemento especializado (el hijo) al general (el padre) como se muestra en la figura 5.1.

Figura 5.1: Símbolo para expresar la generalización.

- Asociación: Es una relación estructural entre dos elementos. Opcionalmente puede también acabar en punta de flecha para indicar la dirección de navegabilidad o incluir ciertos adornos que aumentan su expresividad como se muestra en la figura 5.2.

Figura 5.2: Símbolo para expresar la asociación.

- Realización: Es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Es la relación que hay entre una especificación y su implementación ver figura figura 5.3.

Figura 5.3: Símbolo para expresar la realización.

- Dependencia: Es una relación semántica entre dos elementos, en la cual un elemento requiere la presencia de otro para su implementación o funcionamiento. Un cambio en el elemento requerido (el independiente) puede afectar al del otro elemento (el dependiente).

Se representa como una línea discontinua entre el elemento dependiente, y el independiente acabada en punta de flecha que señala al elemento independiente (esto es, el elemento dependiente indica el elemento del que depende) ver figura figura 5.4.

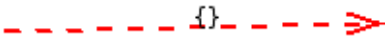


Figura 5.4: Símbolo para expresar la dependencia.

5.3. Diagramas

Un sistema se modela a través de vistas que permiten entender como será implementado un sistema, este modelado depende de la complejidad del sistema a implementar, por lo que las vistas que se usan dependen de la complejidad del sistema y del nivel de detalle que se necesita. Por lo que en términos del modelado de vistas UML ofrece los siguientes diagramas:

1. Diagrama de casos de uso: Muestra a los actores, casos de uso y sus interacciones, el caso de uso representa la funcionalidad del sistema o un subsistema, también muestra la generalización entre actores y la generalización, inclusión y extensión entre los casos de uso.

2. Diagrama de clases: Muestra la estructura estática del modelo, en particular, la definición de las clases y tipos; su estructura interna y su relación con otras clases o tipos.

   Diagramas de comportamiento (modelan cómo se comporta el sistema).

3. Diagrama de estados: Estos son usados para describir el comportamiento de una instancia de un elemento del modelo, tal como un objeto o una interacción. Especialmente esta describe la posible secuencia de estados de una acción a través de la cual la instancia del elemento puede pasar, como resultado de eventos discretos.

4. Diagrama de actividades: Este es una variación del diagrama de estados, en el cual se representa el desempeño de la acción o subactividades, y las transiciones que pueden ser disparadas.

   Diagramas de interacción

5. Diagrama de secuencia: Representa las interacciones entre objetos, usando un conjunto de mensajes.

6. Diagrama de colaboración: El comportamiento es representado por un conjunto de instancias, que intercambian estímulos, para interactuar con el objetivo de realizar una tarea.

   Diagramas de implementación (muestran aspectos relacionados con la implementación).

7. Diagrama de componentes: Muestra la dependencia entre los componentes de software, con sus especificaciones.

8. Diagrama de despliegue.

5.4. Diagrama de casos de uso

Los diagramas de casos de uso representan la funcionalidad del sistema tal como la ven los usuarios. En los diagramas de casos de uso se muestran las diferentes formas posibles de utilización de un sistema. Los diagramas de casos de uso permiten visualizar el comportamiento de un sistema, de forma que los usuarios puedan comprender cómo se utiliza ese elemento y de forma que los desarrolladores puedan implementarlo. En un diagrama de casos de uso importa qué hace el sistema (qué proporciona), no cómo lo hace. Los elementos principales que aparecen en los diagramas de casos de uso son:

- Actores.
- Casos de uso.
- Relaciones.

Los diagramas de casos de uso son una herramienta de comunicación efectiva entre clientes y desarrolladores. Ayudan a capturar los requerimientos del sistema (que será una cuestión básica para todo el desarrollo posterior), para validar el diseño, para obtener los casos de prueba (el sistema obtenido debe proporcionar todas las funcionalidades especificadas en los casos de uso) e incluso como base para la documentación de usuario.

Los diagramas de caso de uso se centran en la interacción entre los usuarios y los sistemas, y no en las actividades internas que tengan lugar en dicho sistema. Son una herramienta muy general que permite especificar las reglas de un negocio. La creación de los casos de uso necesitan de la participación de los usuarios (expertos en el tema a modelar) para su construcción y validación, además de analistas. La creación de los diagramas es un proceso de descubrimiento, que suele requerir varias sesiones.

En síntesis podemos decir que los diagramas de casos de uso:

- Representan una forma estructurada, pero informal, de expresar los requerimientos funcionales.
- Son fáciles de entender, tanto por el cliente como por el desarrollador.
- Son un mecanismo para obtener el consenso en la visión del sistema y sus objetivos entre clientes y desarrolladores.

5.4.1. Actor

Un actor es un rol o conjunto homogéneo de roles que un usuario (persona o máquina) desempeña respecto al sistema. Son por tanto, agentes externos al sistema y que interactúan con él. Gráficamente, un actor se simboliza como se muestra en la figura 5.5. Un actor representa una clase, a la cual pueden pertenecer uno o más elementos (o de forma equivalente, instancias). Por ejemplo, instancias del actor cliente serían: José, Luis, Fernando, etc. Hay tres tipos principales de actores: usuarios del sistema, otros sistemas que interactúan con el sistema que está siendo modelado y el tiempo:

- El primer tipo de actor corresponde a personas físicas o usuarios. No se deben usar nom-

Figura 5.5: Símbolo para expresar un Actor.

bres de personas dado que una misma persona puede desempeñar en un momento dado un rol y en otro instante otro rol.

- El segundo tipo de actor es otro sistema. Nuestro sistema puede interactuar con otros sistemas, que son externos y están fuera de nuestro control .
- El tercer tipo de actor es el tiempo. Cuando el paso de un cierto tiempo dispara algún evento en el sistema. El tiempo funge le rol de actor. Por ejemplo, a medianoche el sistema podría realizar un proceso de control del estado del sistema. Debido a que el tiempo está fuera de nuestro control, es un actor.

Representar un actor del tipo sistema externo o tiempo como en la figura 5.5 puede resultar poco claro. Una opción mejor puede ser asignar un estereotipo a los sistemas externos (por ejemplo «sistema») y utilizar un nuevo icono para su representación.

Los actores se pueden clasificar como primarios o secundarios, de pendiendo si participan en las funciones principales del sistema. también es posible clasificarlos como activos o pasivos, dependiendo de si inician los casos de uso o no.

5.4.2. Casos de uso

Un caso de uso representa una funcionalidad que el sistema proporciona, o de forma equivalente un caso de uso muestra una forma en la que alguien podría utilizar el sistema. Un caso de uso representa una secuencia de transacciones en un sistema cuyo objetivo es proporcionar un resultado mensurable de valor para el actor del sistema. Gráficamente, un caso de uso se representa mediante una elipse, con el nombre del caso escrito dentro o debajo como se observa

en la figura 5.6.

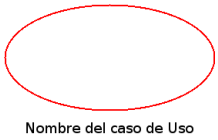


Figura 5.6: Símbolo para expresar un Caso de uso.

Para nombrar los casos de uso se utiliza un verbo o una frase que empiece por un verbo activo, que indique el objetivo o funcionalidad del caso. Los casos de uso pueden describir la funcionalidad a diferentes niveles: sistema, subsistemas y componentes (clases). A cualquiera de estos niveles, los casos de uso definirán el comportamiento del elemento que describen, sin revelar su estructura interna. Se debe precisar que si se utilizan casos de uso para describir clases, los actores que participarán serán en general otras clases o subsistemas. El nivel más importante es el que describe las funcionalidades del sistema. Pertenece al nivel de abstracción del negocio, y es útil tanto para clientes como desarrolladores. Se les denomina casos esenciales o abstractos.

Los casos de uso se deben documentar. Dicha documentación debe contener una descripción del caso de uso y el flujo de eventos del mismo, esta documentación debe incluir la secuencia de acciones, incluyendo las variantes, que un sistema puede ejecutar al ineteractuar, con el objetivo de describir la funcionalidad que proporciona el caso de uso. Opcionalmente, puede incluir precondiciones y/o poscondiciones.

El texto usado en la documentación de los casos de uso debe estar basado en el lenguaje del negocio a modelar. No se recomienda detallar excesivamente ni utilizar pseudocódigo para especificar el flujo de eventos, ya que podría restringir las soluciones adoptadas. Las secciones que podrán aparecer en un caso de uso son:

- Descripción: Describe de forma breve, lo que hace el caso de uso.
- Precondiciones: Son las condiciones que se deben cumplir antes de poder utilizar el caso de uso.
- Poscondiciones: Son las condiciones que se cumplirán después de haber finalizado la eje-

cución del caso de uso.

- Flujo de eventos primario (o base) y alternativo(s). Este describe, paso a paso, lo que sucede en el caso de uso (no cómo sucede). El flujo primario describe la situación normal (debe ser además el primer flujo descrito al crearlo). El flujo o flujos alternativos documentan qué ocurre en situaciones variantes o anormales (por ejemplo, que el cliente no dispusiera de suficiente saldo). Para especificar flujos de eventos se puede además de texto, usar otras técnicas como los diagramas de actividad o de estado.

5.4.3. Relaciones

Hay cuatro tipos de relaciones que pueden aparecer en un diagrama de casos de uso:

1. Relación de comunicación.
2. Relación de inclusión.
3. Relación de extensión.
4. Relación de generalización.

5.4.3.1. Relación de comunicación

La relacióm más común la encontramos entre los actores y los casos de uso esta relación se expresa gráficamente mediante una línea, a la que se puede añadir una punta de flecha para indicar quién inicia la comunicación tal como se observa en la figura 5.7.

Figura 5.7: Relación entre un actor y un caso de uso.

Todos los casos de uso deben ser iniciados por un actor (a excepción de los casos de uso que extienden el comportamiento o están incluidos en otro caso de uso, tal y como se verá más adelante).

5.4.3.2. Relación de inclusión

La relación de inclusión en un caso de uso “A”de caso de uso “B”(esto es, la flecha va de “A”a “B”) indica que el caso de uso “Açontiene el comportamiento indicado por “B”. En el ejemplo de la figura 5.8, se puede observar como el caso de uso “A”y “C”incluyen el comportamiento del caso de uso “B”.

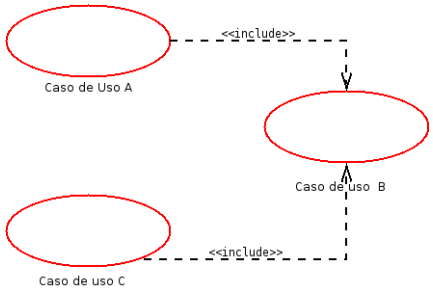


Figura 5.8: Ejemplo de la relación de inclusión.

Las relaciones de inclusión también se pueden utilizar para detallar un caso de uso complejo que proporcione varias subfuncionalidades , de manera que visualmente se puedan apreciar de una forma más fácil. La relación de inclusión indica que un caso de uso va a usar siempre la funcionalidad proporcionada por otro caso.

5.4.3.3. Relación de extensión

Una relación de extensión de un caso de uso “A”a un caso de uso “B”indica que el comportamiento del caso de uso “B”puede ser incrementado con el comportamiento del caso de uso “A”, sujeto a determinadas condiciones, que pueden ser especificadas en la extensión o en el caso de uso como se observa en la figura 5.9

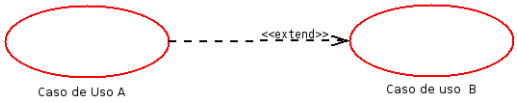


Figura 5.9: Ejemplo de la relación de extensión.

5.4.3.4. Relación de generalización

Esta se puede dar entre actores o entre casos de uso por lo que primero revisaremos entre actores y posteriormente entre casos de uso.

Relación entre actores

La relación de generalización entre actores se utiliza cuando varios actores tienen características comunes. Gráficamente se muestra como una relación de generalización que va del actor especializado al genérico. Una generalización de un actor “B”(descendiente) a un actor “A”(ancestro) indica que una instancia de “B”puede comunicarse con los mismos casos de uso que un actor “A”*como* se observa en la figura 5.10.

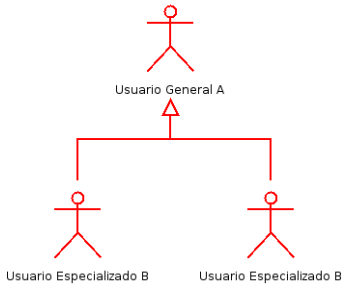


Figura 5.10: Ejemplo de la generalización entre actores.

Se deben de usar las relaciones de generalización entre actores cuando algunos actores tengan en común parte de sus comportamientos.

Relación entre casos de uso

La relación de generalización entre casos de uso se utiliza para indicar que un caso de uso es un refinamiento de otro se diagrama como en la figura 5.11.

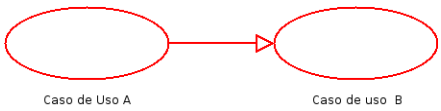


Figura 5.11: Ejemplo de la generalización entre actores.

5.4.3.5. Normas para creación de un diagrama de casos de uso

- No modelar la comunicación entre actores, ya que está fuera del ámbito del sistema.
- No dibujar flechas entre casos de uso, excepto para relaciones de inclusión, extensión o generalización.
- Todo caso de uso debe ser iniciado por un actor (excepto los casos de uso incluidos o las extensiones).
- No se debe representar el flujo de información entre casos de uso. Un caso de uso representa una funcionalidad o un modo de utilizar el sistema, pero no indica el cómo se implementa esa funcionalidad. Por tanto, un caso de uso puede representar la introducción de una información y otro caso de uso el acceso a la misma, sin que exista ninguna comunicación entre ambos casos.
- Es importante que cualquier requerimiento funcional aparezca al menos en un caso de uso, ya que de otro modo no será implementado.
- Los diagramas de casos de uso deben proporcionar fácilmente una visión de lo que el sistema proporciona, tanto a los desarrolladores como a los clientes (para casos de uso abstracto). Se debe usar un número de diagramas adecuado (dependiendo de la complejidad del sistema) con un número de casos de uso razonable y dependiendo del nivel de detalle de dicha vista en particular. Además, los nombres de los casos de uso, actores y descripciones deben ser significativos para el cliente (deben pertenecer al lenguaje del negocio y la organización del cliente).
- Se deben considerar también casos de uso relacionados con el mantenimiento del sistema.
- Los diagramas de casos de uso sólo pueden capturar requerimientos funcionales, y a un

nivel relativamente alto.

5.5. Diagrama de Clases

Los diagramas de clases se utilizan para mostrar la estructura estática del sistema modelado. Pueden contener clases, interfaces, paquetes, relaciones e incluso instancias, como objetos o enlaces.

5.5.1. Clases

Las clases son los componentes fundamentales de los diagramas de clases. La notación general para una clase es un rectángulo dividido en tres secciones, mostrando la primera sección del rectángulo el nombre de la clase (y opcionalmente otra información que afecte a la clase), la siguiente los atributos y la última las operaciones como se muestra en la figura 5.12.

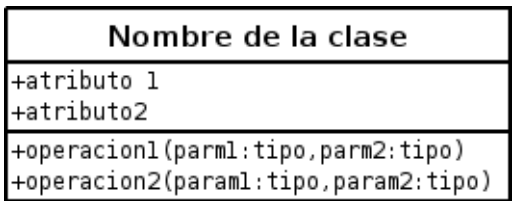


Figura 5.12: Representación gráfica de una clase.

Tanto la sección de los atributos como la de las operaciones pueden no mostrarse, en cuyo caso tampoco se mostrará la línea que las separa. Tampoco es necesario visualizar todos y cada uno de los atributos y operaciones, lo cual puede ser útil para remarcar qué atributos u operaciones son interesantes en una determinada vista. En este caso, conviene añadir puntos suspensivos (...) al final de la sección de la que se hayan suprimido atributos u operaciones.

Otro punto importante es que tanto los atributos como los métodos, deben especificar su visibilidad, dado que puede ser: pública, protegida o privada.

5.5.1.1. Nombre de la clase

El nombre de la clase se debe poner centrado y en negrita, y empezando por letra mayúscula. En caso de que la clase sea abstracta, el nombre se pondrá además en cursiva. Los nombres de clase deben ser únicos en su entorno, pudiendo coincidir sus nombres si están definidos en paquetes distintos (ya que sería otro entorno). Cuando sea necesario, se puede preceder al nombre de la clase por el nombre del paquete donde se ha definido, como por ejemplo **Utilidades::Visualizados.** En caso de especificar un estereotipo para la clase, éste se pondrá encima del nombre de la clase. También es posible sustituir el rectángulo por un icono asociado a dicho estereotipo o poner dicho icono en el extremo superior derecho de la sección.

Lo estereotipos más comunes son:

- «boundary»: Indica que la clase constituye la frontera o interface entre el sistema modelado y el exterior. Se aplica a formularios, informes, interfaces y similares. Su representación gráfica se muestra a continuación.

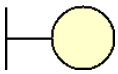


Figura 5.13: Representación gráfica del estereotipo boundary

- «control»: Indica que la clase, es una clase de control que se dedica a labores de coordinación. Estas clases son fáciles de identificar, porque de ellas suelen partir muchos mensajes a otras clases. Su representación gráfica se muestra a continuación.

Figura 5.14: Representación gráfica del estereotipo control

- «entity»: Contienen la información significativa del sistema (entidades) y generalmente son guardadas en almacenamiento externo, esto es, son clases persistentes. Por ejemplo,

en un sistema de almacén los clientes, proveedores y productos serían clases a las que se les aplicaría este estereotipo. Su representación gráfica se muestra a continuación.

Figura 5.15: Representación gráfica del estereotipo entidad

5.5.1.2. Atributos de la clase

En la sección de atributos se muestran todos o parte de los atributos, alineados a la izquierda y utilizando la siguiente sintaxis:

visibilidad nombre[multiplicidad]:tipo=valorInicial cadenaPropiedades

La visibilidad puede ser pública (+), protegida (#), privada (-) o, según el lenguaje, implementación.

El nombre del atributo se escribirá con minúsculas, y a continuación se pondrá entre corchetes su multiplicidad (siendo optativo para el caso habitual de multiplicidad 1). Cuando la multiplicidad es mayor que uno se puede pensar que equivale a definir un array. La multiplicidad en UML se indica como uno o más intervalos enteros (en el caso de múltiples intervalos, se separarán por comas). Cada intervalo tiene el formato límiteInferior..límiteSuperior (cuando el límite inferior es igual que el superior basta con poner simplemente el número). Para indicar un límite indefinido, se utiliza el símbolo asterisco (*), aunque Rose utiliza en este caso la letra ene (n).

Es importante el tipo al que pertenece el atributo este es uno de los existentes en el lenguaje de desarrollo.

5.5.1.3. Operaciones de la clase

En la sección de operaciones se muestran todas o parte de las operaciones que la clase proporciona, alineadas a la izquierda y utilizando la siguiente sintaxis:

visibilidad nombre(listaParámetros):tipoRetorno {***cadenaPropiedades***} En caso de existir parámetros, estos vendrán separados p or comas y siguiendo la siguiente sintaxis para cada parámetro: clase nombre: tipo = valorPorDefecto La visibilidad puede ser pública (+), protegida (#), privada (-) , según el lenguaje de implementación, esto aplica tanto para atributos como para operaciones.

El nombre de la operación se escribirá en minúsculas. En caso de que sea una operación abstracta (recurso utilizado para implementar polimorfismo), se escribirá en cursiva.

En caso de existir parámetros, estos vendrán separados por comas y siguiendo la siguiente sintaxis para cada parámetro: *clase nombre: tipo = valorPorDefecto*

5.5.2. Relaciones

Una relación es una conexión semántica entre elementos, existen cuatro tipos de relaciones enntre clases en UML:

1. Generalización: Es una relación de especialización.

2. Asociación: Es una relación estructural. Existen dos subtipos, la agregación y la composición.

3. Realización: Es una relación contractual, en la cual un clasificador especifica un contrato (por ejemplo, un interface) que otro clasificador garantiza que cumplirá.

4. Dependencia: Es una relación de uso.

5.5.2.1. Generalización.

La generalización es una relación taxonómica entre el objeto más general (el padre o superclase) y el más específico (el hijo o clase descendiente). Gráficamente, se especifica como una línea acabada en flecha vacía que va del elemento más particular (el hijo) al más general (el padre):

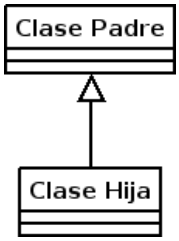


Figura 5.16: Ejemplo de herencia

La generalización se utiliza para modelar la herencia en los lenguajes orientados a objetos. La relación de generalización también puede darse entre paquetes, indicando que un paquete es una especialización de otro. Los paquetes descendientes heredan los elementos públicos y protegidos del paquete ancestro, redefiniendo generalmente algunas clases y definiendo otras nuevas. Por ejemplo:

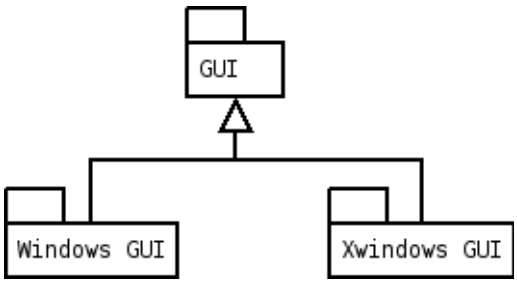


Figura 5.17: Generalización entre paquetes

5.5.2.2. Asociación.

Las asociaciones modelan relaciones estructurales entre clases. Una asociación se representa gráficamente como una línea que conecta las clases relacionadas:

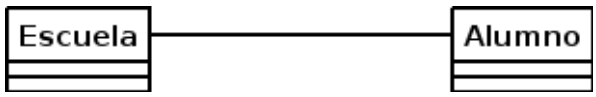


Figura 5.18: Asociación bidireccional entre clases

Esta asociación significa que la clase Escuela puede acceder a los atributos y operaciones públicas de la clase Alumno y que de forma similar, la clase alumno puede acceder a los atributos y operaciones públicas de la clase Escuela. También se puede restringir la navegabilidad de la asociación, añadiendo una flecha que indique el sentido de dicha asociación:

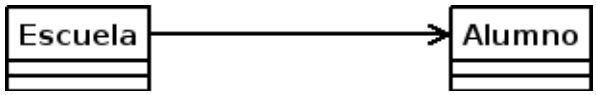


Figura 5.19: Asociación unidireccional entre clases

En este caso, la Escuela puede acceder a los atributos y operaciones públicas de la clase Alumno, pero Alumno no puede acceder a Escuela.

Una clase puede relacionarse consigo misma, es decir, pueden modelarse asociaciones reflexivas:

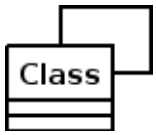


Figura 5.20: Asociación reflexiva

5.5.2.3. Agregación y composición

Estas relaciones son un tipo especial de asociación. La agregación se representa como una asociación con un rombo vacío en el lado de la clase.

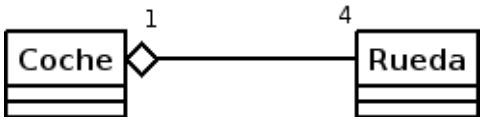


Figura 5.21: Agregación

Trataremos de explicar la agregación con un ejemplo: Pensemos en un coche, el cual tiene cuatro ruedas. y esto puede ser modelado como una agregación, porque las ruedas pueden existir antes que el coche como entidad propia o incluso después (de destruir el carro pueden quedar sus ruedas) e incluso se puede cambiar alguna de las ruedas de un coche a otro.

La composición es mas fuerte que la agregación, porque en la composición define en que momento vive el objeto asociado y solo durante su periodo de vida este existirá. La composición se representa como una asociación con un rombo relleno del lado de la clase.

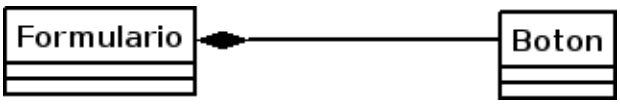


Figura 5.22: Composición

A continuación se revisará otro ejemplo: Un formulario que puede contener cero o más botones. Un botón sólo puede existir dentro de un formulario, por lo que la vida del botón estará restringida a la vida del formulario que lo posee. Un botón pertenece a uno y sólo un formulario. Mientras exista el formulario existirá el botón, en cuanto muera el formulario, morirá el botón.

Tanto en la relación de agregación como en la composición, a la línea que las une se le puede agregar una flecha para ser más claro en la dirección en la que trabaja la asociación.

5.5.2.4. Realización

La realización es una relación semántica, entre clasificadores en la cual un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Generalmente se emplea la realización para especificar una relación entre una interfaz y la clase o componente que implementa las operaciones especificadas en dicha interfaz. Una interfaz es un clasificador que especifica una serie de operaciones, pero que no proporciona ninguna implementación (ni tiene atributos). Es equivalente a una clase abstracta que sólo tiene operaciones abstractas.

Gráficamente, una interfaz se muestra como una clase con el estereotipo «interfaz» y sus operaciones, o en forma resumida como un círculo y debajo el nombre del interfaz:

Figura 5.23: Simbologia para expresar una interfaz

En este caso la interfaz Usuario especifica la operación valida(), para indicar que cierta clase implementará una interfaz, se usa una línea discontinua acabada en flecha vacía que va de la clase que implementa a la interfaz , como se muestra en la figura 5.24.

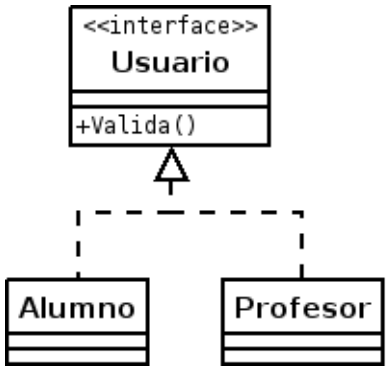


Figura 5.24: Ejemplo de una implementación

Cuando una clase implementa una interfaz, debe implementar todas las operaciones indicadas en dicha interfaz y posiblemente, otras exclusivas de la clase.

5.5.2.5. Dependencia

Una dependencia es una relación entre un elemento dependiente (el cliente) y un elemento independiente (el suministrador del servicio requerido). El elemento dependiente requiere conocer al elemento independiente (suministrador) y que éste esté presente. Las dependencias se usan para modelar relaciones en la cual un cambio en el elemento independiente (el suministrador) puede requerir cambios en el elemento dependiente (el cliente). Es un tipo de relación unidireccional, ya que el elemento dependiente debe conocer al independiente, pero el independiente desconoce la existencia del elemento dependiente. Gráficamente se muestra como una línea discontinua acabada en flecha que va del elemento dependiente al independiente (esto es, el elemento dependiente señala al elemento del que depende). Por ejemplo:

Una relación de dependencia entre un paquete A y un paquete B indica que alguna clase del paquete A tiene una relación unidireccional con alguna clase del paquete B y por tanto, el paquete A necesita tener acceso al paquete B.

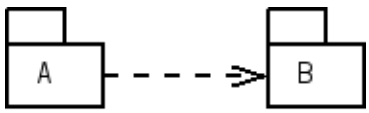


Figura 5.25: Dependencia

5.6. Diagrama de Estados

Un diagrama de estados muestra el ciclo de vida de un objeto (aunque también se puede utilizar para sistemas o subsistemas) desde el momento en que el objeto es creado hasta el momento de su destrucción. Describe todas las secuencias de estados y acciones por las que

un objeto puede pasar durante su vida como reacción a distintos eventos (como pueden ser los mensajes). Un estado es una situación particular en la que se exhibe un comportamiento determinado y que está caracterizado por el valor de uno o más atributos y por las relaciones con otros objetos. El estado en que se encuentre un objeto será por tanto dependiente de su historia anterior.

Los diagramas de estado sólo se suelen realizar para objetos (clases) que tienen un comportamiento dinámico significativo, esto es, para aquellos que poseen un conjunto significativo de estados. Esto sucede con los objetos reactivos, que son aquellos cuyo comportamiento viene dirigido por los eventos que recibe. En un diagrama de estados aparecen uno o más estados relacionados entre sí por transiciones. Las transiciones, en general, serán ocasionadas por algún evento. Por ejemplo. A continuación se muestra un diagrama de estados de un objeto de la clase Curso como se ve en la figura 5.26, este objeto permite la inscripción a un curso, cuando esta en *estado de abierto*, pero cuando finaliza el periódo de inscripciones, cambia el objeto al *estado de cerrado*, si al estar en el estado de cerrado se observa que no tiene alumnos, cambia el estado del objeto a *Cancelado*. Sin embargo si el curso no se cancelo, una vez terminado el periódo de clases pasa al estado de Completado.

5.6.1. Estado

Es una situación en la vida de un objeto durante la cual se satisface alguna condición, se realiza alguna actividad o se espera algún evento. Un estado se representa gráficamente como un rectángulo redondeado como se observa en la figura 5.27

Existen algunas etiquetas de acción reservadas para propósitos especiales y que no pueden ser utilizadas como nombres de eventos. Estas son:

- **entry**: Indica que la tarea se realiza cuando el objeto entra en dicho estado. Se considera una acción no interrumpible.
- **exit**: Indica que la tarea se realiza cuando el objeto abandona el estado. Se considera una acción no interrumpible.

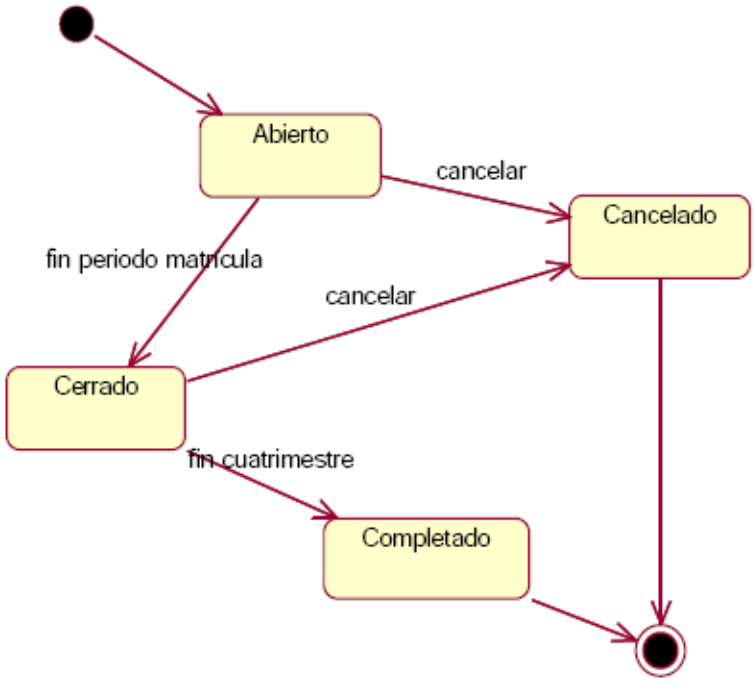


Figura 5.26: Ejemplo de un diagrama de estados

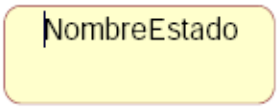


Figura 5.27: Representación gráfica de un estado.

- **do**: Indica que la tarea se realiza mientras el objeto permanezca en el estado en que se encuentra o hasta que termine dicha tarea. Se considera una acción interrumpible. Dicho de otra forma, cuando un objeto está en un estado que contiene una acción etiquetada con do, realiza dicha acción (después de realizar la acción de entrada si la hubiera) hasta que ésta termina o hasta que se abandona dicho estado por la llegada de un evento adecuado.

5.6.2. Estados Especiales

5.6.2.1. Estado inicial

Es un pseudoestado que representa el estado en que se encuentra un objeto cuando es creado (inicio de la vida del mismo). Debe aparecer obligatoriamente uno y sólo uno en todo diagrama de estados. Gráficamente se representa como en la figura 5.28.

Figura 5.28: Representación gráfica de un estado inicial.

5.6.2.2. Estado final

Es un pseudoestado que indica la destrucción del objeto (finalización de la vida del mismo). Es opcional y se pueden añadir varios estados de fin para simplificar el diagrama y evitar el cruce de líneas. Gráficamente se representa como en la figura 5.29.

Figura 5.29: Representación gráfica de un estado final.

5.6.3. Transición

Una transición es el paso de un estado a otro. Se muestra gráficamente como una flecha que va del estado origen al estado destino como se observa en la figura 5.30: Este movimiento puede ser reflexivo, es decir, el estado origen y el estado destino pueden coincidir.

Figura 5.30: Representación gráfica de un estado final.

5.7. Diagrama de Actividades

Los diagramas de actividad permiten modelar el comportamiento de un sistema o alguno de sus elementos, mostrando la secuencia de actividades o pasos que tienen lugar para la obtención de un resultado o la consecución de un determinado objetivo. Opcionalmente, permite mostrar los flujos de información (objetos) producidos como resultado de una actividad y que serán utilizados posiblemente como entrada por la actividad siguiente.

Los diagramas de actividad son útiles para describir flujos de evento en los casos de uso, las actividades que tienen lugar entre un conjunto de objetos (clases) y las operaciones o métodos de las clases. El elemento fundamental de los diagramas de actividad son las actividades. Una actividad representa la ejecución de una tarea o misión en un flujo de trabajo o la ejecución de una sentencia en un procedimiento, dependiendo del elemento cuyo comportamiento se esté modelando.

Una transición se dispara cuando termina de realizarse la actividad de la que parte. También es posible añadir condiciones de guarda a las transiciones. Esto permite realizar decisiones o bifurcaciones basadas en las condiciones de las guardas. Se utiliza un rombo para simbolizar el punto de decisión y el origen de las bifurcaciones.

En los diagramas de actividad se puede mostrar la concurrencia y la sincronización utilizando barras de sincronización, que simbolizan uniones y divisiones en el flujo de control. Obeservemos el siguiente ejemplo figura 5.31. Donde se puede observar la secuencia de actividades para freir unos huevos, primero se consiguen los ingredientes, y una vez que se tienen todos los ingredientes, se puede poner a calentar el aceite y mientras este se calienta, se pueden realizar otras

actividades tal como cascar los huevos y batirlos, sin embargo uno de los flujos puede terminar primero por lo que se usa una barra de sincronización, antes de pasar a freir los huevos.

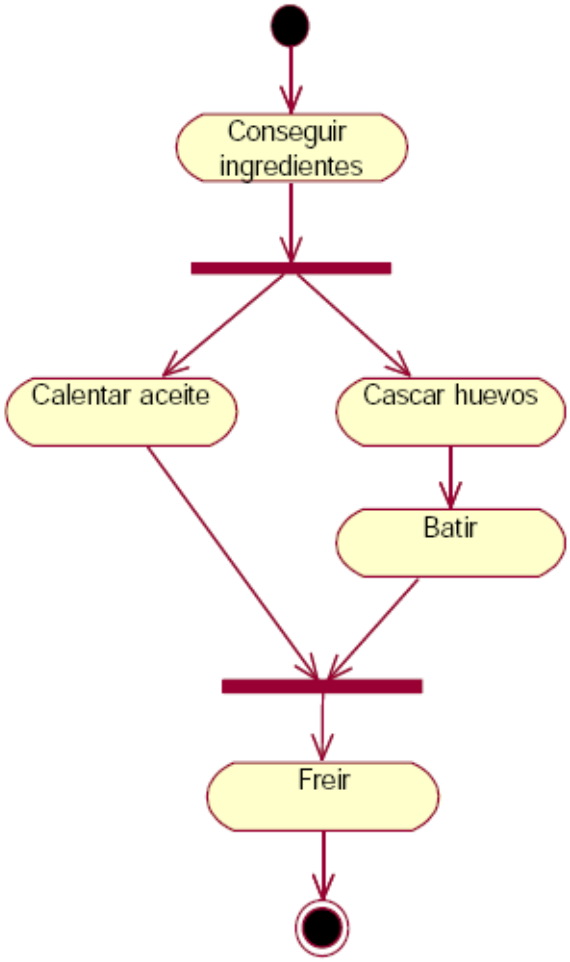


Figura 5.31: Ejemplo de un diagrama de actividades

Como se puede ver, la transición que sigue a la actividad Conseguir ingredientes se divide por medio de una barra de sincronización en dos transiciones que desembocan en las actividades Calentar aceite y Cascar huevos, que pueden ser realizadas de forma concurrente. Por otro lado, la transición que desemboca en la actividad Freir parte de una barra de sincronización a la cual llegan dos transiciones. Este es un punto de sincronización que indica que no se disparará la

transición saliente de la barra hasta que no se hayan disparado las transiciones entrantes (esto es, hasta que no hayan concluido las actividades de las transiciones entrantes: Calentar aceite y Cascar huevos).

5.8. Diagrama de Secuencia

Un diagrama de secuencia es un tipo de diagrama de interacción en el cual se destaca el tiempo: los mensajes entre objetos vienen ordenados explícitamente por el instante en que se envían. Consta de dos ejes. Generalmente, el eje vertical es el eje del tiempo, transcurriendo éste de arriba a abajo. En el otro eje se muestran los objetos que participan en la interacción, siendo el primero de ellos el actor que inicia la ejecución de la secuencia modelada. De cada objeto parte una línea discon tinua, llamada línea de la vida, que representa la vida del objeto durante la interacción. Si el objeto existe durante toda la interacción, éste aparecerá en el eje horizontal y su línea llegará hasta el final del diagrama de secuencia. Los mensajes parten de la línea de vida del objeto que lo envía hasta la línea de vida del objeto al que va destinado (excepto en el caso del mensaje de creación, como se verá más adelante). Cada mensaje lleva un número de secuencia creciente con el tiempo y el nombre de la operación requerida, así como posibles argumentos que pueden utilizarse como valores de entrada y/o salida. Usualmente, no se especifica una graduación en el eje del tiempo, aunque podría hacerse para interacciones que modelen escenarios en tiempo real.

En la figura 5.32 se observan los elementos mas importantes de un diagrama de secuencia:

- Usuario: Es un actor previamente definido que interactura con el sistema.
- Objetos: El diagrama de secuencias trabaja con objetos, es importante definir que nombre tendra el objeto, así como a que clase pertenece.
- Flechas con línea continuas: Esta es para indicar la comunicación entre los objetos, la flecha nace de la linea de vida de un objeto, y va hasta la línea de vida de otro objeto, esta flecha debe tener en la parte superior, el nombre del metodo que se esta invocando,

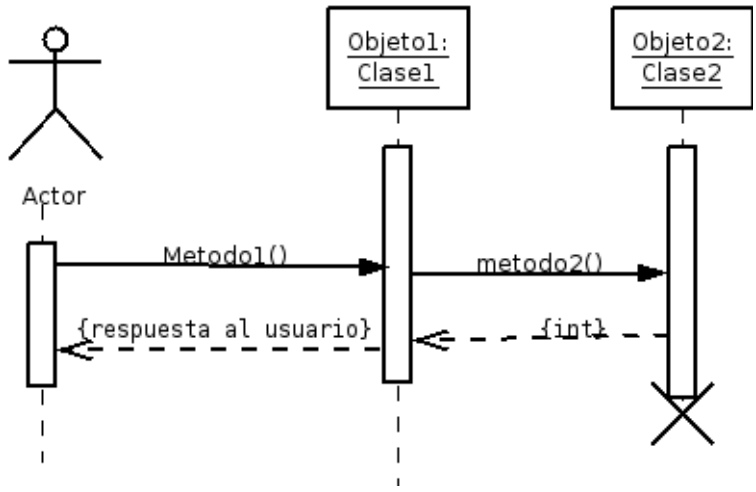


Figura 5.32: Ejemplo de un diagrama de secuencia

así como sus parametros del método.

- Flecha con línea discontinua: Es para indicar la respuesta del metodo de un objeto invocado.
- Una cruz en la línea de vida del objeto: Es para indicar que termino la vida del objeto.
- Linea de vida: Indica en que periódo estuvo vivo el objeto durante la secuencia de actividades.

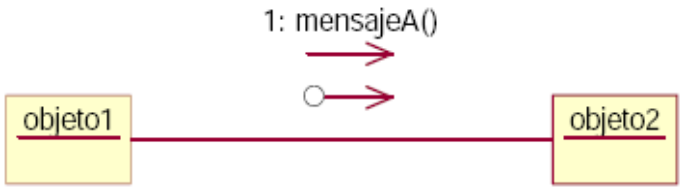
5.9. Diagrama de Colaboración

Un diagrama de colaboración es un diagrama que muestra una interacción en el cual se destaca la organización estructural de los objetos que envían y reciben mensajes. Un diagrama de colaboración muestra los objetos que intervienen en una relación, los mensajes que se intercambian y las relaciones de comunicación que hay entre ellos (para que un objeto pueda comunicarse con otro debe existir algún tipo de enlace o liga entre ellos).

Hay una diferencia significativa entre los diagramas de secuencia y los de colaboración, y es que en los de colaboración se pueden añadir explícitamente flujos de datos. Esto se hace

mostrando al lado del mensaje afectado un circulo al que se añade una flecha indicando la dirección del flujo. Si la dirección del flujo coincide con la del mensaje, se trata de un argumento de entrada. Si no coincide, se trata de un valor de retorno.

Como ejemplo, aquí se muestra un flujo de datos correspondientes a argumentos de entrada del mensajeA que objeto1 envía a objeto2:



5.10. Diagrama de Componentes

Un diagrama de componentes se utiliza para modelar los componentes de un sistema y mostrar las dependencias entre ellos. Un componente representa un módulo de software con un interfaz bien definido. Ejemplos de componentes son el código fuente, código binario, ejecutable, DLL, fichero de inicialización (.ini), fichero de registro (.log), tablas, documentos, mapas de bits de los iconos de una aplicación, etc. Los componentes pueden existir en tiempo de compilación, de enlace o de ejecución o incluso en varios o todos esos tiempos. Los componentes son equivalentes a tipos, siendo sus instancias las que aparecerán en el diagrama de despliegue. Hay una gran diferencia entre clases y componentes. Las clases son “componentes lógicos”, mientras que los componentes tienen significado físico. Antes de poder generar código, las clases deben ser mapeadas a componentes (esto es, las clases residen o son implementadas en componentes), y estos componentes serán los que contengan su código fuente.

La notación estándar de un componente es:

Dentro del componente se especifica su nombre y/o su tipo y, opcionalmente, una cadena de propiedades que indiquen, por ejemplo, su versión, autor, estado, etc. También se puede mostrar

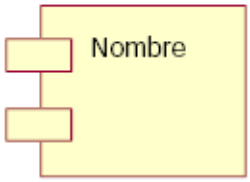


Figura 5.33: Representación gráfica de un componente

dentro de un componente las clases o paquetes que residan en él, o incluso otros componentes.

Los componentes deben tener interfaces bien definidas. Esto permite que los componentes puedan ser reemplazados (por ejemplo, por componentes más eficaces) y reutilizados fácilmente, posibilitando la construcción de componentes complejos utilizando componentes más simples.

Los diagramas de componentes permiten mostrar tanto dependencias en tiempo de compilación y enlazado, como en tiempo de ejecución. Incluso permiten mostrar dependencias entre varias versiones de componentes.

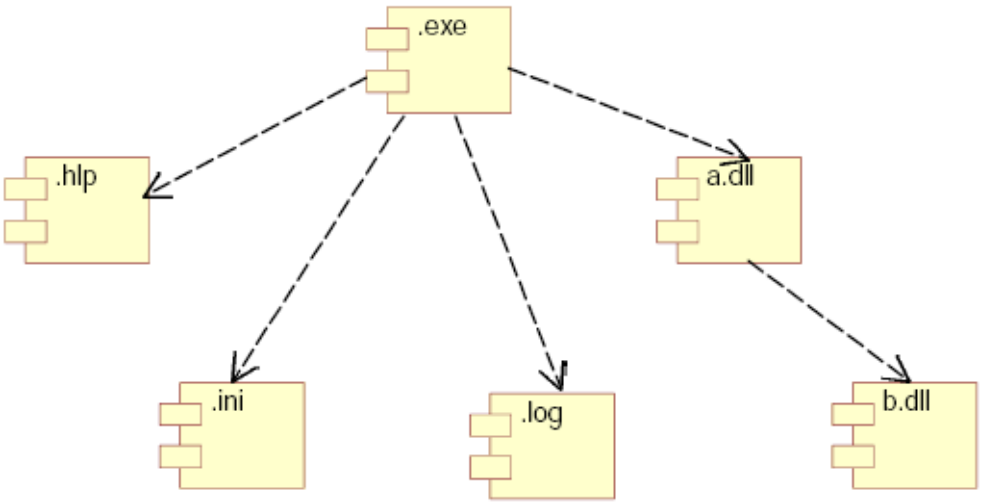


Figura 5.34: Representación gráfica de diagrama de componentes

Capítulo 6

Caso práctico

El caso de estudio que se desarrolló, fue el diseño de un Sistema de Inscripción Virtual (SIV), que permitirá a los alumnos del Departamento de Ingeniería Eléctrica (DIE) del CINVESTAV-IPN inscribirse a los cursos de manera remota mediante el uso de la Internet. Este diseño produjo como resultado un documento que especifica el diseño del SIV (descrito en el apéndice A). Este documento se basa en la Tesis titulada “Análisis Comparativo de Técnicas, Metodologías y Herramientas de Ingeniería de Requerimientos”. Para la etapa de diseño proponemos es este capítulo una guía de diseño para un sistema orientado a objetos.

6.1. Guía para el diseño del sistema

Esta guía tiene como base el proceso de análisis y diseño propuesto por el RUP así como por las etapas que considera el diseño según Ian Somerville[24].

En esta guía se propone las siguientes 7 etapas:

1. Creación de la arquitectura.
2. Revisión de requerimientos.
3. Diseño de los subsistemas.

4. Especificación de las interfaz.

5. Diseño de los casos de uso.

6. Diseño de la Estructura de Datos.

7. Diseño de algoritmos.

6.1.1. Creación de la arquitectura

En este trabajo se propone que se diseñe la arquitectura considerando lo siguiente:

- *La arquitectura en el nivel de negocios*. Esta describe los módulos principales del sistema, con la perspectiva del cliente.

- *La arquitectura en el nivel de aplicación*. Describe como esta organizada la estructura interna del Sistema.

- *La arquitectura Operacional*. Describe los diferentes subsistemas con que interactúa el Sistema para poder operar.

- *La arquitectura en un nivel Tecnológico*. Describe el tipo de hardware y software que se necesitan para la instalación de la aplicación.

6.1.1.1. La arquitectura en el nivel de negocios

Esta etapa es importante por que se identifican los principales subsistemas que componen al sistema así como a los usuarios principales de cada subsistema. Los subsistemas que componen al sistema se pueden representar a través de un diagrama de bloques, donde cada bloque simboliza un subsistema. Los bloques van unidos con flechas que indican el flujo de información, fluyendo de un subsistema a otro. Cada bloque debe ser especificado indicando su funcionalidad. Los bloques pueden agruparse permitiendo expresar la arquitectura del sistema en capas. En el

caso del SIV se manejaron 3 capas. Una para las vistas, otra que controla el nivel de negocios y finalmente una tercera para la manipulación de la base de datos.

Esto ofrece las siguientes ventajas:

- Se disminuye el acoplamiento e incrementa la cohesión.
- Cuando se realiza una modificación en alguna de las capas no afecta a las demás capas.
- Facilita el mantenimiento de los sistemas así como su evolución.
- Se pueda incrementar la funcionalidad con el simple hecho de agregar más capas.

6.1.1.2. La arquitectura en el nivel de aplicación

Habiendo definido los subsistemas y su funcionalidad, a continuación se presenta la funcionalidad de cada subsistema usando diagramas de casos de uso. Los casos de uso son identificados a partir del conjunto requerimientos del sistema. Un requerimiento puede ser satisfecho por uno o varios casos de uso y un caso de uso puede satisfacer uno o varios requerimientos.

Los requerimientos funcionales describen la funcionalidad del sistema, mientras que los casos de uso describen la secuencia de pasos para que el sistema proporcione cierta funcionalidad. Por lo tanto debe existir cierta relación entre los casos de uso y los requerimientos funcionales. Si cada requerimiento funcional se satisface por uno o varios casos de uso y todos los requerimientos funcionales están relacionados con algún caso de uso, estaremos garantizando que todos los requerimientos funcionales recolectados, serán implementados en el sistema.

La arquitectura de un subsistema se puede presentar usando un diagrama de casos de uso, debido a que este modela las relaciones que existen entre los casos de uso y la interacción con los usuarios. Por tal razón , si se sabe que casos de uso conforman a cada subsistema se puede modelar la arquitectura de cada subsistema y por lo tanto se puede modelar todo el sistema usando los casos de uso.

6.1.1.3. La arquitectura Operacional

En esta etapa se identifica a cada uno de los usuarios (actores) del sistema. Esta tarea puede ser sencilla, si ya se identificaron los posibles casos de uso del sistema.

Una vez identificados los usuarios (actores) del sistema, hay que identificar las posibles opciones de cada actor dentro del sistema (su perfil operacional). Dicho con otras palabras ¿Cual es el menú al que tienen derecho cada actor dependiendo su perfil operacional?.

Esta identificación ayuda a mejorar el nivel de dominio del problema y mejora la comunicación con el cliente. Habiendo definido la funcionalidad y la composición de cada subsistema es posible entonces identificar los perfiles operacionales de cada usuario. De esta forma pordrá verificar que los subsistemas presten la funcionalidad que se espera de ellos y será posible identificar desde la perspectiva de los usuarios que requerimientos no se recolectaron.

El perfil operacional (o el ‘menú con las opciones que tiene cada usuario) se puede representar como se muestra en la figura 6.1.

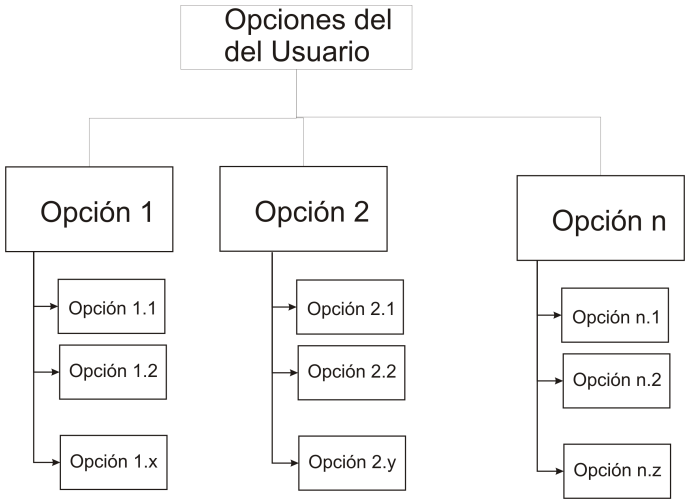


Figura 6.1: Menú de las opciones a las que puede acceder cierto usuario.

6.1.1.4. La arquitectura en un nivel Tecnológico

Esta etapa tiene que ver con la infraestructura y con la factibilidad de la implantación de un sistema, por lo que en esta etapa se definen las características de la infraestructura a utilizar, del equipo de cómputo en el que se instalará el sistema y los dispositivos adicionales que se necesitarán para la operación del sistema. En esta etapa también es importante definir el software que se ocupará para la instalación, el desarrollo, el diseño y la implementación del sistema.

6.1.2. Revisión de requerimientos

Anteriormente se comentó que cada uno de los requerimientos funcionales deben relacionarse con uno o varios casos de uso y que algunos casos de uso pueden relacionados con uno o varios requerimientos como se muestra en la figura 6.2.

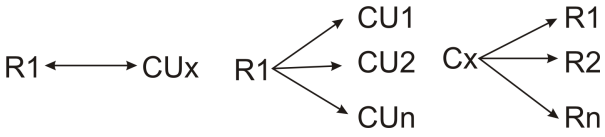


Figura 6.2: Relación de Requerimientos y Casos de Uso

En la revisión de los requerimientos se propone que los requerimientos funcionales sean reordenados por subsistemas y por módulos. Esto con el objetivo de poder generar una tabla que muestre con que casos de uso se podrá construir cada subsistema.Esto con el fin de verificar que todos los requerimientos funcionales han sido relacionados con un caso de uso y por lo tanto poder garantizar que todos los requerimientos funcionales recolectados serán finalmente implementados.

Subsistema 1	R_{x_1}	CU_{x_1}
	R_{y_1} , R_{y_2}, ..., R_{y_n}	CU_{y_1}
	R_{z_1}	CU_{z_1},CU_{z_2},...,CU_{z_n}

6.1.3. Diseño de los subsistemas

Para el diseño del sistema se toma cada subsistema y se realizan las siguientes actividades:

- Se definen las interfaces del subsistema.
- Se hace una descripción de la funcionalidad general del subsistema, así como los datos que necesita para su operación.
- Se crea un diagrama de casos de uso del subsistema.
- Se crea una tabla donde se indican todos los requerimientos que satisface el subsistema, así como el mapeo a los caso de uso.

6.1.4. Especificación de las interfaces

Las interfaces deben estar definidas considerando dos opciones:

- Las interfaces entre subsistemas. En esta se definen los mecanismos de comunicación que ofrece el subsistema con respecto a su entorno.
- Las interfaces de los usuarios con el sistema. En esta se propone generan todas las vistas del subsistema, por usuario o por su perfil operacional, sin que las vistas ofrezcan algún tipo de funcionalidad. Además es muy ilustrativo proporcionar un diagrama de navegación entre las vistas del usuario.

  El diseño de las interfaces del sistema con los usuarios debe planearse antes de implementar el sistema, ya que así los usuarios del sistema podrán evaluar si el sistema ofrecerá, a nivel de vistas la funcionalidad esperada.

6.1.5. Diseño de los casos de uso

Dado que el subsistema se está construyendo con base en los casos de uso, es importante indicar como estos se construyen.

El diseño de los casos de uso debe tener una descripción de su funcionalidad específica y una lista de pasos con los que resuelve cada caso de uso (para lo cual se suguieren entre 7 ó 12 pasos).

En caso de incluir o extender su funcionalidad de otro caso de uso, solo es necesario considerar la funcionalidad que proporciona el caso de uso. Se propone ocupar un diagrama de clases para indicar con que clases se implementará el caso de uso, sin perder de vista que debe de existir un diagrama general de clases del subsistema o módulo. Cada clase debe documentarse indicando su estereotipo, su funcionalidad, y que hace cada métodos. Una vez realizado el diagrama de clases, es necesario indicar la instanciación de objetos y sus relaciones por lo que esto se puede obtener con un diagrama de secuencia o colaboración.

6.1.6. Diseño de la Estructura de Datos

En caso de que el sistema ocupe una base de datos, se puede usar ocupando alguna técnica de diseño de bases de datos como el modelo entidad-relación, tomando la información de los diagramas de clases. En caso de no utilizar una base de datos y ocupar alguna estructua de datos especifica, también se puede obtener información del diagrama de clases.

6.1.7. Diseño de algoritmos

El diseño de algoritmos solo se recomienda usar para métodos que sean realmente complejos, y dado que un método es procedural este puede ser diseñado usando un diagrama de flujo o un diagrama de estados. Se sabe que los diagramas de flujo no son propios del diseño orientado a objetos sin embargo la mezcla de paradigmas de diseño puede generar diferentes opciones de diseño.

Capítulo 7

Conclusiones

El diseño de sistemas es una etapa fundamental en la creación de un sistema. De ésta etapa dependen la implementación, las pruebas, integración operación y mantenimiento de un sistema. Esta etapa es importante para poder realizar un sistema dentro de un tiempo y un presupuesto estimado, por lo que un buen diseño impactará en el desarrollo, costo y funcionalidad de los sistemas computacionales.

Por tal razón en esta tesis se ha propuesto una guía simplificada de diseño, la cual se se uso en el caso práctico del apéndice A, y se observo que se puede disminuir la cantidad de documentación. Además los documentos presentan una estructura sencilla, lo que ayuda a su uso e interpretación por parte de los diferentes usuarios involucrados en el desarrollo del sistema.

La guía de diseño que se propone en el capítulo 6 permite la creación de software con calidad porque satisface los siguientes criterios de calidad: rastreabilidad, mantenibilidad y usabilidad.

- *La rastreabilidad*. Es proporcionada porque la documentación generada permite identificar el proceso por el cual, los requerimientos funcionales fueron implementados. Esto es porque existe un mapeo que permite saber como los requerimientos se convirtieron en casos de uso y posteriormente con que clases se implemento y con que mensajes se comunican estas clases para satisfacer el requerimiento.
- *La mantenibilidad*. Si existe la documentación de las clases con las que fue implementado

un subsistema y estas clases tiene baja cohesion. Se favorece el poderle dar mantenimiento a los subsistemas.

- *La Usabilidad*. Dado que en la etapa de diseño de la arquitectura, los clientes ayudaron a definir las vistas con las que operaría el sistema. Este no será ajeno a ellos cuando el sistema sea instalado. Lo cual ayudará a reducir el tiempo de aprendizaje de este.

La guía de diseño favorece el poder estimar el costo y el tiempo que puede tener un sistema. Porque, ofrece a través de la definición de la arquitectura, el entender junto con el cliente el problema a resolver y garantiza que los requerimientos funcionales recolectados sean implementados.

El uso de un lenguaje estándar como UML disminuye la ambigüedad del diseño, en el (SIV) se ocupo básicamente los diagramas de: casos de uso, clases, secuencias y colaboración. Dado que con estos se puede indicar como implementar el sistema, y proporcionan la arquitectura de cada subsistema.

La mezcla de metodologías puede llegar a ser una excelente opción en el diseño de sistemas, obteniendo lo mejor de cada metodología se podría generarse un espacio amplio de soluciones.

7.1. Trabajo Futuro

Proponer métricas formales, que sustenten los resultados observados en la guía de diseño propuesta. Incrementar a la guía de diseño, ciertos procesos que ayuden a incrementar los criterios de calidad. Combinar esta guía con el uso de patrones de diseño, para favorecer el rehusó de soluciones.

Bibliografía

[1] How do you define software architecture? . http://www.sei.cmu.edu/architecture/definitions.html.

[2] ”ieee recommended practice for architectural description of software-intensive systems”. IEEE Std 1471-2000.

[3] Software engineering – product quality. http://www.iso.org/iso/en/CombinedQueryR CombinedQueryResult?queryString=iso+9126.

[4] Len. Bass. *Software architecture in practice*. Addison-Wesley, 1998.

[5] Allend H. Dutoit Bernard Bruegge. *Ingeniería de Software Orientado a Objetos*. Prentice Hall, 2002.

[6] S. Bodoff, D. Green, K. Haase, E. Jendrock, and B. Stearns. *The J2EE Tutorial*. Addison-Wesley, 2002.

[7] Paul C. Clements. A survey of architecture description languages. *International Workshop on Software Specification and Design*, 1996.

[8] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns*. Addison-Wesley, 2002.

[9] Booch Grady. *Object-Oriented Analysis and Design with Applications*. Addison Wesley, 1994.

[10] Ivar Jacobson. *Object-oriented software engineering : a use case driven approach*. Addison Wesley, 1992.

[11] Philippe Kruchten. *The Rational Unified Process An Introduction*. Addison Wesley, 2000.

[12] Theo Mandel. *The Elements of User Interface Design*. Wiley, 1997.

[13] Lisandra V. Manzoni and Roberto T. Price. Identifying extensions required by rup (rational unified process) to comply with cmm (capability maturity model) levels 2 and 3. *IEEE Transactions on Software engineering, Vol. 29, No. 2*, (2), February 2003.

[14] David Garlan Mary Shaw. *Software architecture : pespectives on an emerging discipline*. Prentice Hall, 1996.

[15] OMG,doc, http://cgi.omg.org/cgi-bin/doc?ad/01-03-08. *Software Process Engineering Metamodel (SPEM)*, april 2001.

[16] Peninsula School of Computing and Information Technology Monash University,McMahons Road, Frankston, Vic 3199, Australia. *A Comprehensive Interface Definition Framework for Software Components*, Montréal, P.Q., Canada, 2001. Jun Han.

[17] Philippe Kruchten Per Kroll. *Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison Wesley, 2003.

[18] PShari Lawrence Pfleeger. *Ingeniería de Software: Teoría y Práctica. Madrid*. Prentice-Hall, 2000.

[19] Roger S. Pressman. *Ingeniería de Software un enfoque práctico*. Mc Graw Hill, 2003.

[20] Proceedings of the 14th international symposium on Systems synthesis (ISSS '01). *Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures*, Montréal, P.Q., Canada, september 2001. Alex Nicolau, Nikil Dutt, Prabhat Mishra.

[21] James Rumbaugh. *Object-oriented modeling and design*. Prentice Hall, 1991.

[22] Mary Shaw. Comparing architectural desgn styles. *IEEE Computer Society*, (6), November 1994.

[23] Mary Shaw and David Garlan. An introduction to software architecture. *CMU Software Engineering Institute Technical Report*, January 1994.

[24] Ian Sommerville. *Ingenieria de Software*. Adison Wesley, 2002.

[25] Alfredo Weitzenfeld. *Ingenieria de Software Orientada a Objtos con UML, Jave e Internet*. Thomson, 2004.

[26] Liping Zhao and Elizabeth A. Kendall. Role modelling for component design. *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, Janue 2000.