U N I V E R S I D A D A U T Ó N O M A

M E T R O P O L I T A N A

Fundamentos de Ingeniería de Software

Dra. María del Carmen Gómez Fuentes

Dr. Jorge Cervantes Ojeda

Dr. Pedro Pablo González Pérez

2019

Editores
María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda
Pedro Pablo González Pérez

Departamento de Matemáticas Aplicadas y Sistemas.
División de Ciencias Naturales e Ingeniería
Universidad Autónoma Metropolitana, Unidad Cuajimalpa

Editada por:
UNIVERSIDAD AUTONOMA METROPOLITANA
Prolongación Canal de Miramontes 3855,
Quinto Piso, Col. Ex Hacienda de San Juan de Dios,
Del. Tlalpan, C.P. 14787, México D.F.

Fundamentos de Ingeniería de Software

Primera edición 2019

ISBN: 978-607-28-1659-6

Impreso en México
Impreso por Litoprocess S. A. de C.V.
Calzada San Francisco Cuautlalpan 102A
53569 Naucalpan Estado de México

Contenido

1 Introducción

1.1 Presentación

"Fundamentos de ingeniería de software" es la primera Unidad de Enseñanza Aprendizaje (UEA) en la que los estudiantes de Ingeniería en Computación de la Universidad Autónoma Metropolitana Unidad Cuajimalpa (UAM-C) tienen un acercamiento con la ingeniería de software. Este primer acercamiento sirve para comprender su importancia y para familiarizar al estudiante con los temas medulares de su carrera, que son: los modelos de desarrollo de software, el proceso de requerimientos, el diseño, la codificación, las pruebas y la calidad del software. En cursos posteriores se estudia cada uno de estos temas con mayor profundidad. El programa de estudios de Fundamentos de ingeniería de software no incluye el tema de la gestión de la ingeniería de software, por lo tanto, no lo incluimos, ya que éste tema se estudia en la UEA Administración de Proyectos. Cabe mencionar que aquí se presentan por primera vez en un libro de texto los Diagramas de Transición entre Interfaces de Usuario (DTIU), que son una aportación de los autores [Gómez y Cervantes, 2013]. Como una actividad integradora del conocimiento, se incluyen cuestionarios con los temas para cada capítulo. Los casos de estudio de este libro de texto sirven para ilustrar algunos de los modelos de desarrollo de software, y son también ejemplos sencillos de especificación de requerimientos, diseño, pruebas y aplicación de reglas de codificación. Finalmente, en el último capítulo se presenta una introducción a las metodologías ágiles de desarrollo de software.

1.2 El programa de estudios de "Fundamentos de Ingeniería de Software"

A continuación, presentamos el contenido sintético del programa de estudios de la UEA "Fundamentos de Ingenieria de Software".

CONTENIDO SINTÉTICO

1. El proceso de desarrollo de software como el conjunto estructurado de actividades requeridas para elaborar un sistema.
   1.1. Especificación de Requerimientos.
   1.2. Conceptos y principios del diseño. Introducción a UML.
   1.3. Codificación. Estándares y procedimientos de programación. El paradigma de la Programación Estructurada. El paradigma orientado a objetos.
   1.4. Conceptos de la calidad del software
   1.5. Pruebas y mantenimiento del software.

2. Modelos de clases y objetos
   2.1. Diagramas de clases. Clases, atributos y métodos.
   2.2. Relaciones de clases. Herencia, agregación, asociación y dependencia.
   2.3. Tipos de clases. Abstracta, estáticas y otras.
   2.4. Paquetes de clases.
   2.5. Diagramas de objetos. Objetos y relaciones entre ellos.

3. Modelos de desarrollo de software
   3.1. El modelo en cascada y sus ciclos de vida: Pura. Con fases solapadas. Con subproyectos. Con reducción de riesgos
   3.2. El modelo evolutivo y sus ciclos de vida: Espiral. Entrega por etapas o incremental. Entrega evolutiva o iterativo: Diseño por planificación. Cascada en V.
   3.3. Minimización de desarrollos y sus ciclos de vida: Componentes Reutilizables. Diseño por herramientas.

4. Metodologías ágiles de desarrollo de software
   4.1. Concepto de la metodología ágil.
   4.2. Programación extrema.
   4.3. SCRUM.
   4.4. Desarrollo dirigido por pruebas.
   4.5. Desarrollo dirigido por Características.

5. Casos de estudio.

Como se puede apreciar en el Contenido, este libro cubre el 100% del programa de estudios de la UEA "Fundamentos de ingeniería de software" y, los casos de estudio son material para desarrollar en el laboratorio. Para comprender mejor este libro, es recomendable que el estudiante tenga conocimientos de Programación Orientada a Objetos.

1.3 Objetivos

1.3.1 Objetivo General

Comprender los conceptos fundamentales de la ingeniería del software, y la importancia de su aplicación, para la creación de productos de software de calidad.

1.3.2 Objetivos específicos:

- Comprender el proceso de desarrollo de software (ciclo de vida) como un conjunto estructurado de las actividades requeridas para realizar un sistema y el modelo de desarrollo de software como la representación abstracta de este proceso.
- Comprender las actividades necesarias para elaborar un sistema de software.
- Ubicar los diferentes ciclos de vida dentro de los modelos de desarrollo de software.
- Explicar el proceso de construcción de un sistema de software a través de los modelos de desarrollo de la ingeniería del software.

1.4 Conocimientos previos

Es deseable que el lector tenga nociones de programación estructurada y de programación orientada a objetos para que tenga una mejor comprensión de este libro.

2 Introducción a la ingeniería del software

2.1 Objetivos específicos del capítulo

- Comprender qué es la ingeniería de software y sus principios fundamentales.
- Comprender la importancia de aplicar la ingeniería de software en el desarrollo de sistemas computacionales.
- Comprender la diferencia entre proceso y modelo de desarrollo de software.
- Considerar el concepto de software con base en sus características, componentes y áreas de aplicación.

2.2 ¿Qué es la ingeniería de software?

La ingeniería de software se ha definido por varios autores. Según Ian Sommerville, considerado uno de los padres de la ingeniería de software, la ingeniería de software "es una disciplina de la ingeniería que comprende todos los aspectos de la producción del software" [Somerville, 2004].

La IEEE (The Institute of Electrical and Electronics Engineers: el Instituto de Ingenieros Electricistas y en Electrónica), es la asociación internacional más grande del mundo formada por profesionales de las nuevas tecnologías, como ingenieros electricistas, electrónicos, en sistemas y en telecomunicaciones.

La IEEE define a la ingeniería de software como "la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software".

Una de las definiciones más interesantes es la de Bohem (1976): "ingeniería de software es la aplicación práctica del conocimiento científico al diseño y construcción de programas de computadora y a la documentación asociada requerida para desarrollar, operar y mantenerlos. Se conoce también como desarrollo de software o producción de software". Definimos la ingeniería de software como:

Una disciplina en la que se aplican técnicas y principios de forma sistemática en el desarrollo de sistemas de software para garantizar su calidad.

2.3 Guía del cuerpo de conocimiento de la ingeniería de software

El softwareEBOK[1] [Bourque y Fairley, 2014] (*Guide to the software engineering body of knowledge*: Guía del cuerpo de conocimiento de la ingeniería de software) es un documento editado por la IEEE y describe el conocimiento que existe en la disciplina de ingeniería de software. Divide a la ingeniería de software en 12 áreas:

1. Requerimientos
2. Diseño
3. Construcción
4. Pruebas
5. Mantenimiento
6. Gestión de la configuración
7. Gestión de la ingeniería de software (Administración de Proyectos)
8. Procesos software

[1] https://www.computer.org/education/bodies-of-knowledge/software-engineering

9. Métodos y herramientas
10. Calidad
11. Medición
12. Seguridad

A lo largo de este libro estudiaremos una introducción a los temas básicos de la ingeniería de software, que son: requerimientos, diseño, construcción (codificación y pruebas), gestión de la configuración, calidad y mantenimiento.

2.4 ¿Por qué es importante la ingeniería de software?

[2]Normalmente, los clientes que mandan construir un sistema de software lo quieren lo más pronto posible (o antes). Es por esto que para "acabar más rápido" existe la tentación de comenzar a codificar y probar sin hacer antes un buen análisis de cuáles son los requerimientos del cliente ni un diseño.

Si no se tienen claros los requerimientos del cliente, entonces lo más probable es que el sistema que se desarrolle no cumpla con sus necesidades. Cuando no se cuenta con un diseño, se cometen todo tipo de errores y es muy difícil encontrarlos y resolverlos de forma adecuada. Lo más probable es que el tiempo invertido para que el sistema funcione haya sido mayor al que se hubiera invertido aplicando ingeniería de software. Como consecuencia, los desarrolladores quedan frustrados y exhaustos, y el cliente difícilmente quedará satisfecho. Además, al final se entrega un producto de mala calidad al que difícilmente se le podrá dar un buen mantenimiento.

A lo largo de la reciente historia del desarrollo de software (comparada con la de otras ingenierías) se han tenido experiencias muy amargas:

- Millones de dólares invertidos en sistemas que nunca llegaron a funcionar como se esperaba.
- Millones de dólares pagados a empresas de software que jamás pudieron entregar el sistema.

[2] Imagen de http://korolevatc.rusedu.net/gallery/3094/TQ1LTg0YT.jpg

- Errores en el diseño de sistemas que causaron daños enormes e irreparables (aviones que se han caído, problemas financieros en los bancos, etc.)

Para que las empresas de software pudieran sobrevivir, buscaron varias alternativas y llevaron a cabo procesos que siguen una metodología. La práctica ha demostrado que una metodología ayuda a minimizar los grandes problemas que se presentan al desarrollar sistemas complejos. La ingeniería de software es el resultado de la acumulación de las diferentes metodologías que han mostrado, en la práctica, ser útiles para mejorar la calidad de los sistemas desarrollados.

La experiencia dice que, si en un proyecto grande no se aplica ingeniería de software, éste fracasará. La probabilidad de que un proyecto sea exitoso es mucho mayor cuando se aplica la ingeniería de software.

2.5 Principios de la ingeniería de software

Aunque la ingeniería de software no tiene principios fundamentales universalmente reconocidos [Bourque et. al, 2002], se han hecho varios intentos por establecerlos [Boehm, 1983; Davis, 1995]. "Fundamentals of software Engineering " [Ghezzi et. al, 2002] es un reconocido libro de ingeniería de software (más de 1650 citas a la fecha) que propone 7 principios que constituyen *las propiedades deseables* durante el desarrollo de un sistema de software. A continuación, se explica cada uno de estos principios.

- *Rigor y formalidad*. Si la documentación del sistema es ambigua o inconsistente será difícil encontrar los defectos y saber dónde hacer cambios. Mientras más rigor y formalidad en la documentación y código, el sistema será más confiable, verificable y mantenible.

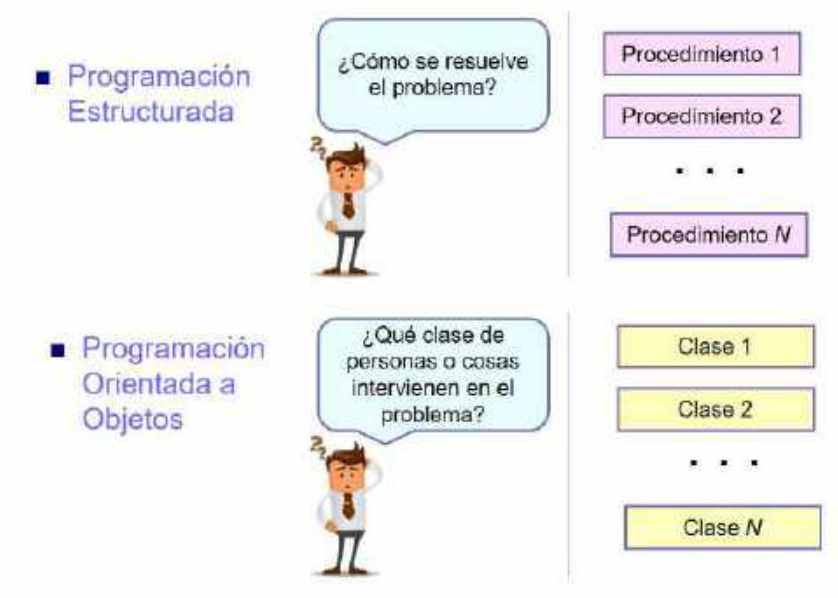


Figura 2-1: Modularidad en la programación estructurada y en la programación orientada a objetos

- *Modularidad*. Para resolver un problema complejo de desarrollo de software, conviene separarlo en partes más pequeñas que se puedan diseñar, desarrollar, probar y modificar de manera sencilla y lo más independientemente posible del resto de la aplicación. A cada una de estas partes se les llama módulos.
  En la programación estructurada los módulos son procedimientos y funciones que, en conjunto, proporcionan la funcionalidad del sistema, mientras que en la programación orientada a objetos los módulos están formados por las clases que son parte del problema. Como se observa en la Figura 2-1, los módulos en programación estructurada están orientados a procesar una parte de la solución del problema, mientras que en programación orientada a objetos están enfocados a determinar cuáles son las propiedades de las entidades que intervienen en el problema.

- *Abstracción.* Abstraer significa *obtener la esencia* al identificar o percibir el problema. Con la abstracción se extraen características comunes a partir de ejemplos específicos. En el ejemplo de la Figura 2-2 se observa que, de las figuras geométricas con diferentes tamaños y excentricidades, se hace la abstracción de que todas ellas son, en esencia, elipses porque todas tienen en común las características de una elipse.

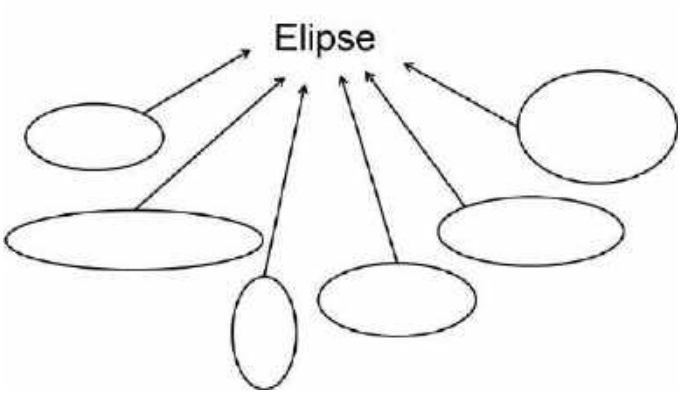


Figura 2-2: Ejemplo de abstracción

- *Anticipación al cambio.* El software sufre cambios constantemente. Las principales causas de los cambios son: la necesidad de eliminar defectos que no fueron detectados antes de liberar la aplicación, el surgimiento de nuevos requerimientos o cambios en los requerimientos existentes. Se requiere un esfuerzo especial en las fases iniciales del desarrollo de un proyecto de software para anticipar cómo y dónde es probable que se den los cambios. Con la *gestión de la configuración* se administran las diferentes versiones del software de forma controlada.

- *Generalidad*. Generalizar es buscar un problema que sea lo más general posible en lugar de tener varias soluciones especializadas. Para resolver un problema se debe buscar un problema más general que posiblemente esté oculto tras el problema original. El problema original puede reutilizarse. Así, un mismo módulo puede ser invocado desde más de un punto en la aplicación en lugar de tener varias soluciones en módulos especializados.

- *Incrementalidad*. La experiencia ha demostrado que los requerimientos del usuario van cambiando o se van definiendo mejor mientras se desarrolla el producto. Bajo estas circunstancias, es común que se hagan varias entregas parciales del sistema cada vez más completas. Cuando se construye una aplicación en forma incremental, los pasos intermedios pueden ser *prototipos* del producto final que permiten ir teniendo retroalimentación del usuario y descubrir y acordar así cuáles son sus verdaderos requerimientos. O bien, se puede comenzar por un núcleo del sistema con la *funcionalidad* más importante, y posteriormente ir entregando las demás funcionalidades.

- *Separación de intereses*. Bajo este principio se separan diferentes aspectos de un problema para concentrarse en un aspecto y después atender los otros. Por ejemplo, si se requiere que un programa sea *correcto* y *eficiente*, nos concentramos primero en una solución que resuelva correctamente el problema y posteriormente lo modificamos para lograr mayor eficiencia.

2.6 Características, componentes y áreas de aplicación del software

El software es el elemento lógico de un sistema basado en computadora, a diferencia del hardware que es el elemento físico. El software se crea a través de una metodología de desarrollo con fases y actividades claramente definidas, y no se fabrica en un sentido clásico como la mayoría de los productos. Hoy en día, el desarrollo de software implica el ensamblaje, reusabilidad y extensibilidad de componentes existentes.

Los componentes del software son los programas ejecutables y los datos existentes en una computadora. Las áreas de aplicación del software son diversas. En la Figura 2-3 se ilustran algunos ejemplos.

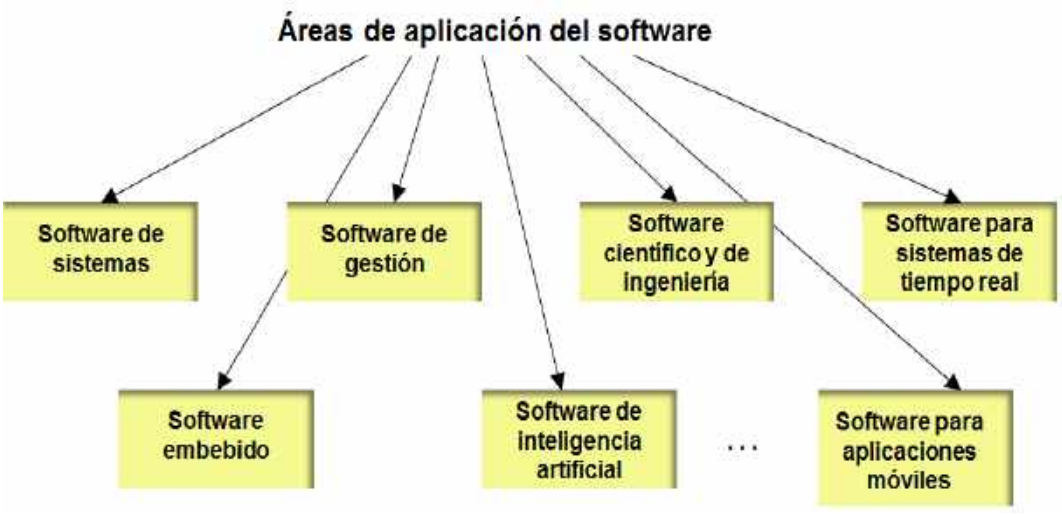


Figura 2-3: Áreas de aplicación del software

2.7 *Proceso* (Ciclo de vida) y *Modelo* de desarrollo de software

2.7.1 Proceso de desarrollo de software

Un **proceso de desarrollo de software** es el conjunto estructurado de las actividades requeridas para construir un sistema. El *proceso de desarrollo de software* se utiliza para mejorar la comprensión del problema a resolver, la comunicación entre los participantes del proyecto y el mantenimiento de un sistema complejo. Las actividades fundamentales de este proceso son: especificación de requerimientos, diseño, codificación, validación (pruebas) y mantenimiento (corrección, adaptación, extensión, mejoras). En la Figura 2-4 se ilustran los aspectos involucrados durante el proceso de desarrollo de software: la aplicación de principios, técnicas y prácticas de la ingeniería, las ciencias de la computación, la gestión de proyectos, el dominio de aplicación y otros campos relacionados.

Figura 2-4: Aspectos involucrados durante el proceso de desarrollo de software

Los métodos indican como construir "técnicamente" el software, y abarcan, entre otras, las siguientes tareas:

- Recolección inicial de requerimientos.
- Planeación y estimación del proyecto.
- Recolección formal de los requerimientos.
- Análisis de los requerimientos del sistema.
- Diseño de las estructuras de datos, arquitectura de los componentes y procedimientos algorítmicos.
- Codificación.

- Prueba.
- Implantación.
- Mantenimiento.

Existen herramientas para semiautomatizar o automatizar diferentes métodos del desarrollo de software, tales como el análisis, diseño, implementación o pruebas. Como ejemplos de herramientas clásicas para el desarrollo de software, podemos mencionar desde las iniciales herramientas CASE (del inglés, *Computer-Aided software Engineering*), hasta los más sofisticados ambientes integrados de desarrollo (del inglés, *Integrated Development Environment*).

Tanto las herramientas CASE como los IDE son aplicaciones informáticas, cuyo objetivo es proporcionar apoyo semiautomatizado o completamente automatizado en cualquiera de los métodos o actividades involucrados en el proceso de desarrollo de software. Las herramientas CASE y los IDE contribuyen significativamente a agilizar el proceso de desarrollo de software, reduciendo el costo del mismo en términos de tiempo, recursos humanos y recursos financieros.

Finalmente, los procedimientos son el pegamento que une los métodos y las herramientas, facilitando un desarrollo racional y oportuno del software de computadora. Los procedimientos definen la secuencia en que se aplican los métodos, los artefactos, herramientas y documentos. Los procedimientos también son los controles que ayudan a asegurar la calidad y coordinar los cambios, y las directrices que permiten evaluar el progreso.

En la Figura 2-5 se ilustran las actividades de la definición, el desarrollo y el mantenimiento del software.

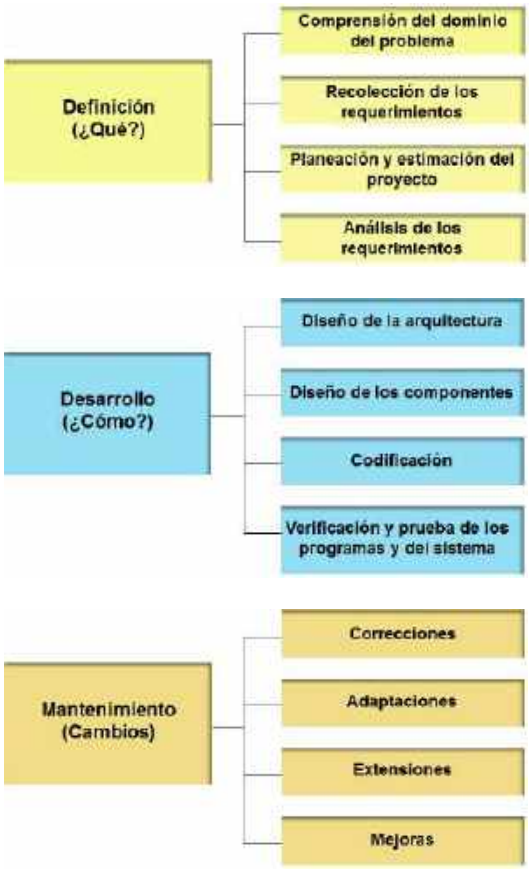


Figura 2-5: Las actividades de la definición, el desarrollo y el mantenimiento del software

Las principales actividades, involucradas en las tres anteriores macro etapas del proceso de desarrollo de software se ilustran en la Figura 2-6.

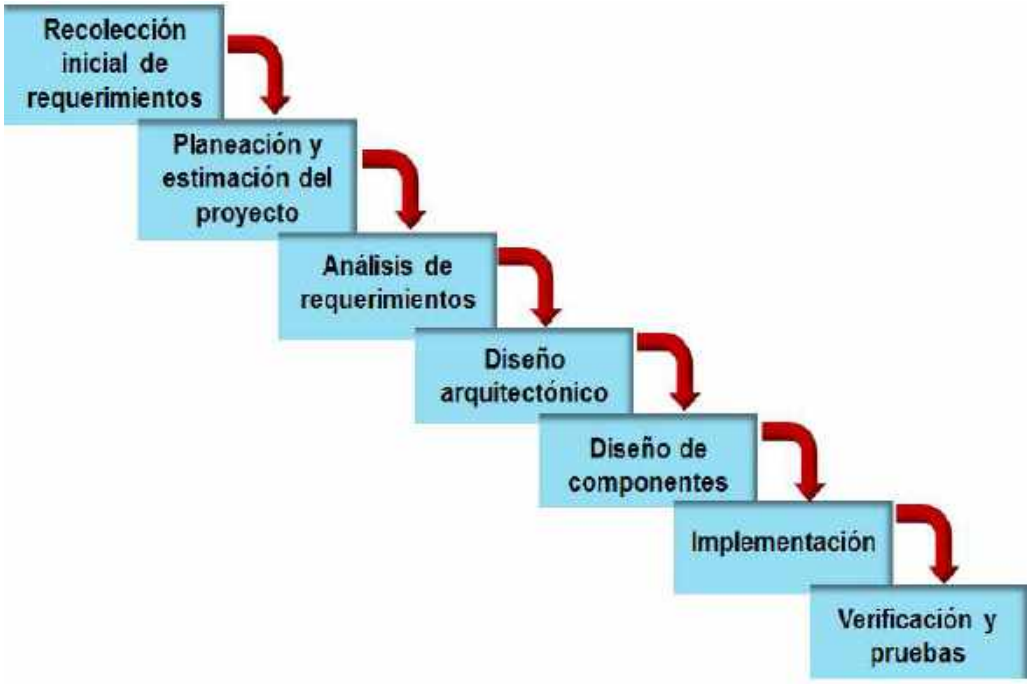


Figura 2-6: Actividades involucrados durante el proceso de desarrollo de software

Para efectos didácticos, las actividades o etapas de la Figura 2-6 se muestran como parte de una secuencia lineal, sin embargo, en dependencia del ciclo de vida adoptado (por ejemplo, prototipado evolutivo, espiral o entrega por etapas), el proceso puede ser cíclico e iterativo, existiendo etapas a las que es posible regresar.

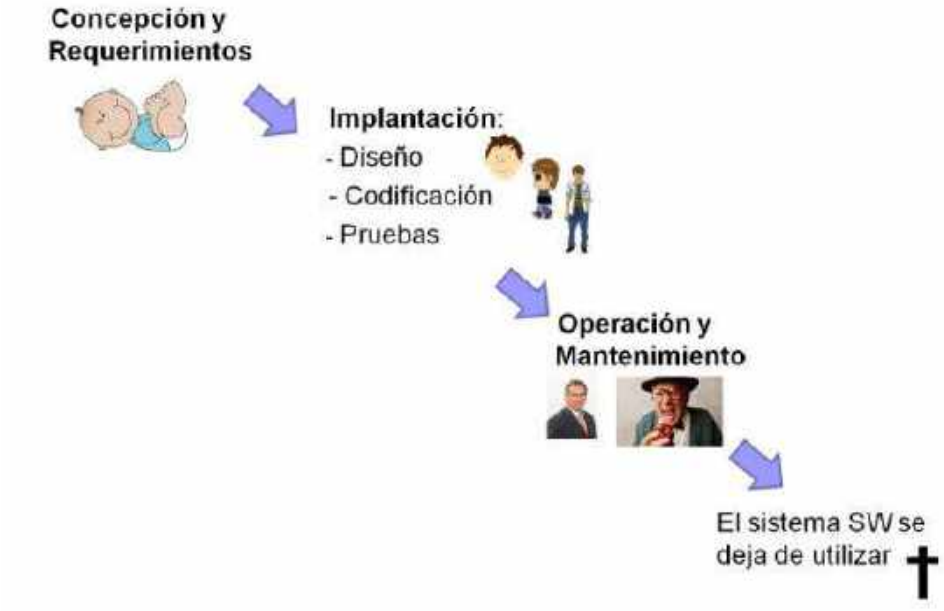


Figura 2-7: El ciclo de vida de un sistema de software

Al proceso de desarrollo de software también se le conoce como *ciclo de vida*. Un ciclo de vida es el proceso que se sigue para construir, entregar y hacer funcionar el software, desde la concepción de la idea del sistema hasta su entrega y el desuso. En la Figura 2-7 se describe el ciclo de vida de un producto de software. Primero nace con la concepción del sistema, su planeación y la especificación de los requerimientos. Luego se lleva a cabo su implantación que consiste en su diseño, codificación y pruebas. Posteriormente el producto se entrega y sigue viviendo durante su operación y mantenimiento. El ciclo de vida del sistema de software termina cuando éste se deja de utilizar.

2.7.2 Modelo de desarrollo de software

Un ***modelo*** *de desarrollo de software* es una representación abstracta del *Proceso de Desarrollo de software*, y determina **el orden en el que se llevan a cabo las actividades del proceso de desarrollo de software.** El *Modelo de desarrollo* es el procedimiento que se sigue durante el proceso de desarrollo de un sistema de software y a éste también se le llama *paradigma del proceso*. El modelo indica el orden de las etapas involucradas en el desarrollo del software y nos proporciona un criterio para comenzar, para continuar a la siguiente etapa y para finalizar.

Las empresas grandes y competitivas trabajan con modelos de desarrollo de software y además con sistemas de calidad que permiten verificar que el desarrollo del software se lleve a cabo adecuadamente.

Existe una gran variedad de paradigmas (modelos) de desarrollo de software. Si los clasificamos por grupos podemos entender las ventajas y desventajas de cada grupo.

Universal	Modelo Abstracto	Modelos Concretos
Modelos Tradicionales (pesados)	Cascada	Pura En V Con fases solapadas Con subproyectos Con reducción de riesgo
	Evolutivos	Espiral Incremental (Entrega por etapas) Iterativo (Entrega evolutiva) Prototipos desechables
	Orientados al reuso (Component – Based)	COTS-based software development Product-line development Architecture-driven component development
	Hibridos	Rational Unified Process (Proceso Unificado)

Figura 2-8: Clasificación de los modelos tradicionales de desarrollo de software

La Figura 2-8 muestra la clasificación de los modelos tradicionales de desarrollo de software, a los que también se les llama *modelos tradicionales o pesados*, mientras que la Figura 2-9 contiene la clasificación de los modelos y metodologías basadas en la administración del cambio (*modelos ágiles*). Como puede observarse, en cada una de estas clasificaciones hay una categoría abstracta, la cual contiene las características comunes presentes en los modelos concretos. En capítulos posteriores se estudian cada uno de los modelos incluidos en las tablas de la Figura 2-8 y la Figura 2-9.

Universal	Modelo Abstracto	Modelos Concretos
Modelos y metodologías basadas en la gestión del cambio (Agiles)	Desarrollo Ágil y modelos con enfoque en la administración	Programación Extrema (XP) Feature-driven development (FDD) SCRUM Test-driven development (TDD) Dynamic Systems Development (DSDM) Crystal Clear Otros
	Principios y metodologías basadas en la práctica	Agile Modeling (AM) Lean Otros

Figura 2-9: Clasificación de los modelos ágiles de desarrollo de software

2.8 Resumen

La ingeniería de software es una disciplina en la que se aplican técnicas y principios de forma sistemática en el desarrollo de sistemas de software para garantizar su calidad, y es el resultado de la acumulación de las diferentes metodologías que han mostrado, en la práctica, ser útiles para mejorar la calidad de los sistemas desarrollados

Un *proceso de desarrollo de software* es el conjunto estructurado de las actividades requeridas para construir un sistema, y se utiliza para mejorar la comprensión del problema a resolver, la comunicación entre los participantes del proyecto y el mantenimiento de un sistema complejo.

Las actividades fundamentales del proceso de desarrollo de software son: especificación de requerimientos, diseño, codificación, validación (pruebas) y mantenimiento (corrección, adaptación, extensión, mejoras).

2.9 Cuestionario

1.- Explica los principios de la ingeniería de software:

- Rigor y formalidad.-.
- Modularidad.-.
- Abstracción.-
- Anticipación al cambio.-
- Generalidad.-
- Incrementalidad.-
- Separación de intereses.-

2.- ¿Qué es un **proceso** de desarrollo de software?

3.- Enlista las actividades fundamentales del proceso de desarrollo de software

4.- ¿Qué es un **modelo** de desarrollo de software?

3 El Proceso de Requerimientos

3.1 Objetivos específicos del capítulo

- Comprender el concepto de requerimiento de un sistema de software.
- Comprender la importancia del proceso de requerimientos.
- Conocer en qué consisten cada una de las actividades del proceso de requerimientos.

3.2 Introducción

La captura, el análisis y la especificación de los requerimientos del sistema es una de las fases más importantes para que el proyecto tenga éxito.

3.2.1 ¿Qué son los requerimientos?

Los requerimientos especifican qué es lo que un sistema de software debe hacer (sus funciones) y sus propiedades esenciales y deseables. Un requerimiento expresa el propósito del sistema sin considerar cómo se va a implantar. En otras palabras, los requerimientos identifican **qué hace** el sistema, mientras que el diseño establece el **cómo lo hace** el sistema.

3.2.2 ¿Para qué sirven los requerimientos?

Los requerimientos pueden servir a tres propósitos:

- Primero, permiten que los desarrolladores expliquen cómo han entendido lo que el cliente espera del sistema.
- Segundo, indican a los desarrolladores qué funcionalidad y qué características debe tener el sistema resultante.
- Tercero, indican qué demostraciones se deben llevar a cabo para convencer al cliente de que el sistema que se le entrega es de hecho lo que había ordenado.

3.3 Tipos de requerimientos

La IEEE divide los requerimientos en ***funcionales*** y ***no funcionales***. Los requerimientos ***funcionales*** describen cómo debe comportarse el sistema ante determinado estímulo. Son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones particulares. En algunos casos, también pueden declarar explícitamente lo que el sistema no debe hacer.

Entonces: *Los requerimientos **funcionales** de un sistema describen lo que el sistema debe hacer*.

Los requerimientos ***no funcionales*** describen una restricción o limitación sobre el sistema. Restringen los servicios o funciones ofrecidas por el sistema. Incluyen restricciones de tiempo, el tipo de proceso de desarrollo a utilizar, fiabilidad, tiempo de respuesta, capacidad de almacenamiento.

Entonces: *Los requerimientos **no funcionales** ponen límites y restricciones al sistema*.

3.4 Características de los requerimientos

Deben ser correctos.- Tanto el cliente como el desarrollador deben revisarlos para asegurar que no contengan errores.

Deben ser consistentes.- Dos requerimientos son inconsistentes cuando es imposible satisfacerlos simultáneamente.

Deben ser realistas.- Todos los requerimientos deben ser revisados para asegurar que sea posible implementarlos en un tiempo razonable.

Deben ser verificables.- Se deben poder preparar pruebas que demuestren que se han cumplido los requerimientos.

3.5 El proceso de Requerimientos

El procesamiento de los requerimientos es un subproceso de la ingeniería de software que, debido a su papel clave en el proceso de desarrollo, se estudia como un proceso por separado. Hay cursos y libros especializados que profundizan en el estudio de cada una de las actividades que se llevan a cabo durante el proceso de requerimientos. Estas actividades se ilustran en la Figura 3-1. En las siguientes subsecciones se explica cada una de ellas.

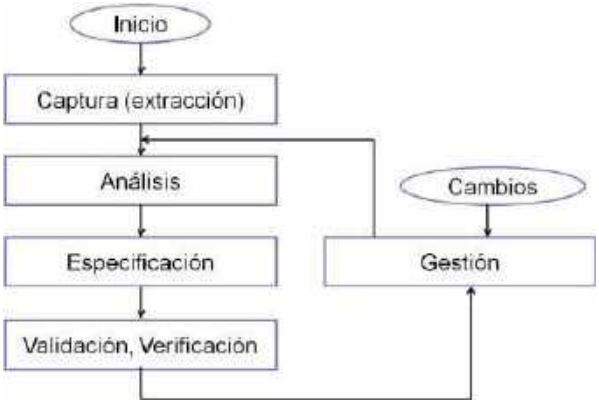


Figura 3-1: El proceso de requerimientos

El proceso de requerimientos es el conjunto de técnicas y procedimientos que nos permiten conocer los elementos necesarios para definir un proyecto de software y, como ya habíamos mencionado, es una de las fases más importantes para el éxito del desarrollo del proyecto.

3.5.1 Captura de los Requerimientos

La captura o extracción de los requerimientos tiene como objetivo principal la comprensión de lo que los clientes y los usuarios esperan que haga el sistema.

Durante la captura se identifican los aspectos clave que el sistema requiere y se descartan los aspectos irrelevantes. Existen diversas técnicas para la extracción de los requerimientos, aquí mencionaremos las más comunes. Una de ellas es la entrevista con el cliente y los usuarios. También está la observación de tareas, en la que se revelan problemas, detalles, estrategias y estructuras de trabajo que son difíciles de captar claramente con las entrevistas. Con el prototipado se va modificando un modelo hasta que éste cumple con las expectativas del cliente. Por último, mencionamos las reuniones JAD (Joint Application Development: Desarrollo Conjunto de Aplicaciones), la cual es una práctica de grupo que se desarrolla durante varios días y en la que participan analistas, usuarios, administradores del sistema y clientes para concretar las necesidades del sistema que se piensa desarrollar.

3.5.2 Análisis de los requerimientos

Durante el análisis de los requerimientos se extraen las características operacionales del software, se indican las interfaces entre usuario y sistema y se establecen las restricciones que debe cumplir el software. También se traducen los requisitos del usuario a requerimientos de software y se clasifican los requerimientos. Además, se detecta si hay conflictos entre los requerimientos y se resuelven estos conflictos. El objetivo del analista de requerimientos es reconocer los elementos básicos de un sistema tal como lo percibe el usuario/cliente.

3.5.3 Especificación de los requerimientos

La Especificación de Requerimientos es un documento que define, de forma completa, precisa y verificable, los requisitos, el diseño y el comportamiento u otras características, de un sistema o componente de un sistema.

Mientras que el análisis de requerimientos proporciona una vía para que los clientes y los desarrolladores lleguen a un acuerdo sobre lo que debe hacer el sistema, la especificación de requerimientos es el producto de este análisis, y proporciona las pautas a seguir por los desarrolladores del sistema.

Para escribir la Especificación de Requerimientos de software se utiliza como guía un estándar, como el de la IEEE-830, el PSS-05 de la Agencia Espacial Europea o la plantilla "Volere" [Robertson y Robertson, 1999]. Estas plantillas proveen con una estructura específica para presentar los requerimientos. Por ejemplo, en la IEEE-830: introducción, descripción general, requerimientos específicos, apéndices e índice.

Para realizar bien el desarrollo de software es esencial tener una especificación completa de los requerimientos. Independientemente de lo bien diseñado o codificado que esté, un sistema pobremente especificado decepcionará al usuario y su mantenimiento será difícil.

La Especificación de Requerimientos suministra al técnico y al cliente, los medios para valorar la calidad del sistema que se entrega ya terminado.

De la experiencia se sabe que el costo de reparar un error se incrementa en un factor de diez [Pfleeger, 2006] de una fase de desarrollo a la siguiente, por lo tanto, la preparación de una Especificación de Requerimientos adecuada, reduce los costos y los riesgos asociados con el desarrollo del proyecto.

3.5.4 Técnicas y métodos del análisis de requerimientos según el Proceso Unificado de Desarrollo de software

En adición a lo antes expuesto sobre el análisis y especificación de requerimientos, las técnicas y métodos proporcionados por UP para esta fase de desarrollo de software constituyen un valioso soporte para enriquecer aún más la especificación de los requerimientos (ver Tabla 3-1).

Técnicas y métodos del análisis de requerimientos según UP	**Breve descripción**
Declaración escrita del alcance del sistema de software: Visión y análisis del negocio	(1) Describe los objetivos y las restricciones de alto nivel, (2) Brinda una descripción del dominio desde una perspectiva de las principales funcionalidades/ requerimientos y objetos identificados, (3) Proporciona elementos para la toma de decisiones.
Diagrama de contexto	Ilustra los principales componentes del sistema de software, sus dependencias de

	otros sistemas o entidades, así como los límites entre el sistema de software y su ambiente.
Modelo de Casos de Uso	Describe los requerimientos funcionales y su relación con los no funcionales.
Glosario	Se refiere a la terminología clave del dominio. Es la definición de los términos más importantes que sobresalen en el análisis del documento de los requerimientos. Estos términos no tienen que corresponder necesariamente con los objetos del dominio.
Especificación Complementaria	Es la descripción de todos los requerimientos funcionales, la cual complementa el modelo de casos de uso. Esta descripción puede ser hecha de forma textual, con tablas, con otros diagramas, etc.
Diagramas de clases	Un diagrama de clases describe los tipos de objetos (clases) que componen el sistema y los diferentes tipos de relaciones estáticas que existen entre éstos.

Tabla 3-1: Técnicas y métodos del Proceso Unificado para el Análisis de Requerimientos

3.5.5 Validación y verificación de los requerimientos

3.5.5.1 Validación de requerimientos

La validación tiene como objetivo detectar defectos en los requerimientos antes de invertir dinero en las siguientes etapas del proceso de desarrollo.

El objetivo principal de la validación de requerimientos es comprobar que éstos realmente definen el sistema que el cliente desea y que lo que describen es lo que el cliente pretende ver en el producto final. Otros objetivos importantes de la validación son: asegurar que los requerimientos estén completos, sean exactos y consistentes.

La validación es importante porque la detección de errores durante el proceso de análisis de requerimientos reduce mucho los costos. Si se detecta un cambio en los requerimientos una vez que el sistema está hecho, los costos son muy altos, ya que significa volver a cambiar el diseño, modificar la implementación del sistema y probarlo nuevamente.

Las principales técnicas de validación de requerimientos son: *revisiones de requerimientos* y *construcción de prototipos*.

Una *revisión de requerimientos* es un proceso manual en la que intervienen tanto el cliente como personal involucrado en el desarrollo del sistema. Los participantes verifican que el documento de requerimientos no presente anomalías ni omisiones.

La *construcción de prototipos* consiste en mostrar un modelo ejecutable del sistema a los usuarios finales y a los clientes, así éstos pueden experimentar con el modelo para ver si cumple con sus necesidades reales.

Con la ***validación de requerimientos*** se contesta a la pregunta: ¿*Se está construyendo lo que pidió el cliente*?

3.5.5.2 Verificación de requerimientos

Una característica esencial de un requerimiento de software es que debe ser posible verificar que el producto final lo satisface. Una tarea importante es planificar cómo *verificar cada requerimiento*. La verificación de requerimientos pretende asegurar que el sistema de software satisfará las necesidades del cliente y se lleva a cabo mediante las *pruebas de aceptación***.**

Las *pruebas de aceptación* sirven para convencer al cliente de que el sistema cumple con lo que él pidió. Debe diseñarse por lo menos una prueba para cada requerimiento. Si un requerimiento no se puede probar significa que el requerimiento está mal hecho y es necesario replantearlo.

Con la ***verificación de requerimientos*** se contesta a la pregunta: ¿*Lo que pidió el cliente, funciona correctamente*?

3.5.6 Gestión de los requerimientos

La *gestión de requerimientos* es el proceso de organizar y llevar a cabo los cambios en los requerimientos con el objetivo de asegurar la consistencia entre los requerimientos y el sistema construido. Abarca todo el ciclo de vida del producto de software, y es una tarea que requiere de una buena cantidad de tiempo y esfuerzo.

Durante la *gestión de requerimientos* se comprenden y controlan los cambios en los requerimientos del sistema.

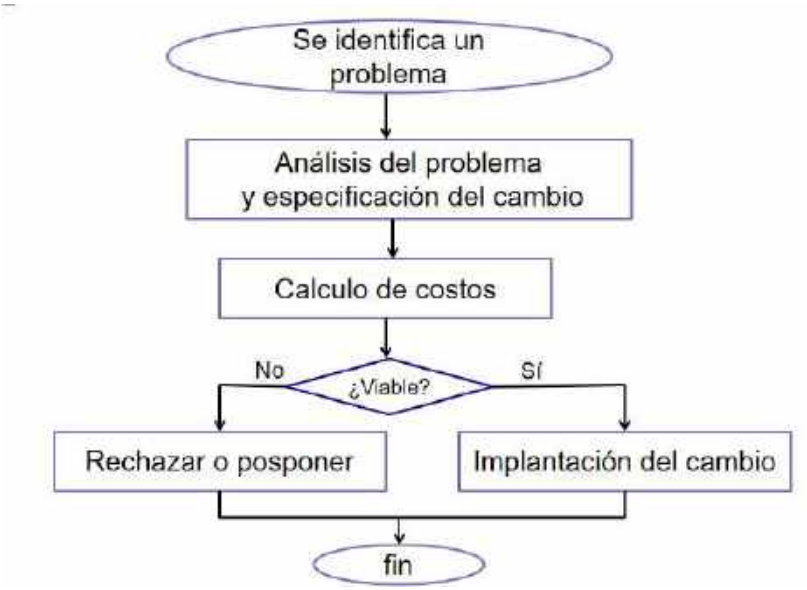


Figura 3-2: Gestión de los requerimientos

Como se observa en la *Figura 3-2*, la gestión de un cambio surge con la identificación de un problema, el cual se analiza y se resuelve con un cambio propuesto. Después se valora el efecto del cambio propuesto para calcular su costo. Este costo se estima tomando en cuenta las modificaciones al documento de requerimientos, y si es el caso, al diseño e implementación del sistema. Al finalizar esta etapa se toma una decisión sobre si se continúa o no con el cambio del requerimiento. En caso afirmativo, se lleva a cabo la implantación del cambio. De lo contrario, se rechaza o se pospone.

Con la finalidad de que todos los cambios propuestos sean tratados de forma consistente y controlada, en la industria de software de calidad se utiliza un *proceso formal de gestión de cambios*, es decir un proceso sistematizado y estándar.

3.6 Resumen

Los requerimientos especifican qué es lo que un sistema de software debe hacer (sus funciones) y sus propiedades esenciales y deseables. Un requerimiento expresa el propósito del sistema sin considerar cómo se va a implantar.

Los requerimientos *funcionales* de un sistema describen lo que el sistema debe hacer, mientras que los requerimientos *no funcionales* ponen límites y restricciones al sistema.

Proceso de obtención de los requerimientos consta de varias etapas. En la primera se hace la extracción de los requerimientos consultando las necesidades del cliente, después se hace un análisis para verificar que cumplan con las características deseables de un requerimiento y después se hace un documento formal al que se le llama “Especificación de Requerimientos”. Posteriormente, las modificaciones a los requerimientos y los nuevos requerimientos se controlan mediante un proceso de gestión.

Existen herramientas de modelado que sirven para ilustrar los requerimientos, una de las que más se utiliza es la de los casos de uso.

3.7 Cuestionario

1.- ¿Qué es un requerimiento de un sistema de software?

2.- ¿Cuál es la diferencia entre los requerimientos funcionales y los no funcionales?

3.- Dibuja el proceso de requerimientos.

4.- ¿Cuál es la diferencia entre el ***Análisis de Requerimientos*** y la ***Especificación de Requerimientos***?

5.- ¿Cuál es el objetivo de la ***validación de requerimientos***?

6.- ¿Para qué sirven las ***pruebas de aceptación***?

7.- Dibuja el proceso de gestión de requerimientos.

4 Modelado de software

4.1 Objetivos específicos del capítulo

- Comprender el concepto de modelo y la utilidad de modelar durante el desarrollo de software.
- Conocer los diagramas UML más utilizados.
- Conocer los Diagramas de Transiciones entre Interfaces de Usuario (DTIU)

4.2 Introducción

4.2.1 Definición de modelo y la utilidad de modelar.

Un modelo es una *representación* de un concepto, de un objeto o de un sistema, que permite explicar algunas propiedades específicas, como, por ejemplo: la forma, el comportamiento, la estructura o la relación de una entidad con otras entidades.

Como se ilustra en la Figura 4-1, cuando un programador hace un modelo mental del sistema que construye es difícil comunicarlo a otros programadores, pues la comunicación verbal no es muy precisa. Incluso se puede esbozar el sistema en un pizarrón o en un pedazo de papel, sin embargo, es muy probable que el concepto mental no sea entendido por otros programadores si no se ponen de acuerdo y usan un mismo lenguaje para describir el modelo mental que cada uno tiene del sistema.

Si el autor de un sistema de software no escribe los modelos que estaban dentro de su cabeza cuando los diseñó, esta información podría perderse para siempre, incluso para él mismo.

[3]

Figura 4-1: La comunicación verbal no es muy precisa (imagen disponible en internet)

Modelamos para comprender mejor el sistema que estamos desarrollando, es decir, para visualizar lo que queremos que haga el sistema, especificar su estructura y comportamiento, y guiar su construcción. Además, modelar ayuda a comunicarse con el cliente o usuario, a explorar posibles soluciones a un problema durante el diseño del sistema, y a documentar las decisiones tomadas durante el desarrollo. Lo anterior es indispensable durante la producción de software de calidad.

4.3 El Lenguaje Unificado de Modelado (UML)

UML (Unified Modelling Language) es el **lenguaje de modelado** de sistemas de software más conocido y utilizado en la actualidad. Se le dice "unificado" porque para crearlo se pusieron de acuerdo los tres máximos exponentes del diseño orientado a objetos:

[3] Imagen de https://www.transients.info/wp-content/uploads/2019/02/david-topi-5.jpg

Grady Booch: desarrolló su propia notación para el análisis y diseño orientado a objetos.

James Rumbaugh: a su propia notación de diseño orientado a objetos le llamó OMT (Object Modeling Technique).

Ivar Jacobson: visionario del análisis y diseño orientado a objetos, creador de los casos de uso.

A mediados de los noventa, los tres empezaron a intercambiar documentos y a trabajar en conjunto. Obtuvieron la versión 1.0 de UML la cual fue un gran avance en el modelado de sistemas orientados a objetos.

OMG (Object Management Group) es la Asociación que establece y administra estándares de tecnologías orientadas a objetos. En 1997 UML se dio a conocer cuando fue aceptado y respaldado por la OMG. Después de pasar por varias revisiones y refinamientos a la fecha, la última versión de UML es la 2.5.

UML es un LENGUAJE DE MODELADO, no es una METODOLOGÍA DE DESARROLLO. En una metodología de desarrollo, cada método se define en términos de un lenguaje de modelado y de un proceso para modelar.

UML se usa como lenguaje gráfico para:

- ✓ visualizar
- ✓ especificar
- ✓ construir
- ✓ documentar

UML es visual, y mediante su sintaxis se modelan distintos aspectos que permiten una mejor interpretación y entendimiento de un problema o de un sistema computacional.

Los elementos UML se utilizan para crear diagramas y no están diseñados exclusivamente para software orientado a objetos. UML se compone de muchos elementos de esquematización que representan las diferentes partes de un sistema de software, es decir, consta de varios tipos de diagrama. Cada diagrama sirve para modelar diferentes partes o puntos de vista de un sistema de software.

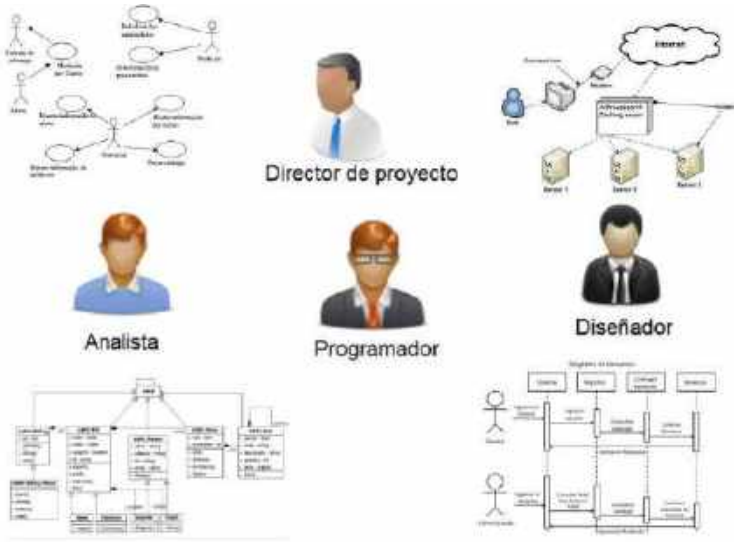


Figura 4-2: UML facilita la comunicación entre los miembros del equipo de desarrollo

Entonces: *UML es un lenguaje para el modelado de varios aspectos del sistema de software*. Como se ilustra en la Figura 4-2, los diagramas UML facilitan la comunicación entre el equipo de desarrollo de software.

4.4 UML en las diferentes etapas del proceso de desarrollo

Entre más complejo es el sistema que se desea crear el uso de UML proporciona más beneficios. Por ejemplo, con un modelo global del sistema es más fácil detectar las dependencias entre los módulos del sistema y las dificultades implícitas para implementarlos. Además, con los diagramas se facilita la detección de problemas en las etapas de análisis y de diseño, reduciendo así los costos.

Los diagramas UML se usan en diferentes etapas a lo largo del proceso de desarrollo de software ya que sirven para complementar la documentación de las diferentes fases de un proceso de desarrollo de software. UML es útil en cualquier tipo de ciclo de vida (Cascada, evolutivo, ágil, etc.).

4.4.1 UML en el documento de Especificación de Requerimientos

Con UML se pueden hacer modelos de un sistema que sean precisos, completos y no ambiguos. Esto facilita la *especificación de requerimientos* de un sistema de software extenso.

4.4.2 UML en el documento de diseño

El documento de diseño puede contener una gran variedad de diagramas para ilustrar: 1) los módulos que componen el sistema, 2) los diferentes estados por los que pasan (si aplica), 3) la manera en la que los módulos se comunican entre sí y 4) la secuencia en la que intervienen para ofrecer cada una de las funcionalidades requeridas.

UML ayuda a visualizar el diseño y a hacerlo más accesible para otros.

4.4.3 UML en la construcción

Es posible mapear los elementos de un modelo UML a un lenguaje de programación tal como Java, C++, Visual Basic y viceversa, esto ha permitido avances como la *generación automática de código* y la *ingeniería inversa*.

4.5 Los diagramas UML

Los diagramas UML se clasifican en dos grupos:

- Los **Diagramas de Estructura** que describen los *elementos que deben existir* en el sistema modelado.
- Los **Diagramas de Comportamiento** que describen *lo que debe suceder* en el sistema modelado.

En la Figura 4-3 se muestran los tipos de diagramas UML que existen.

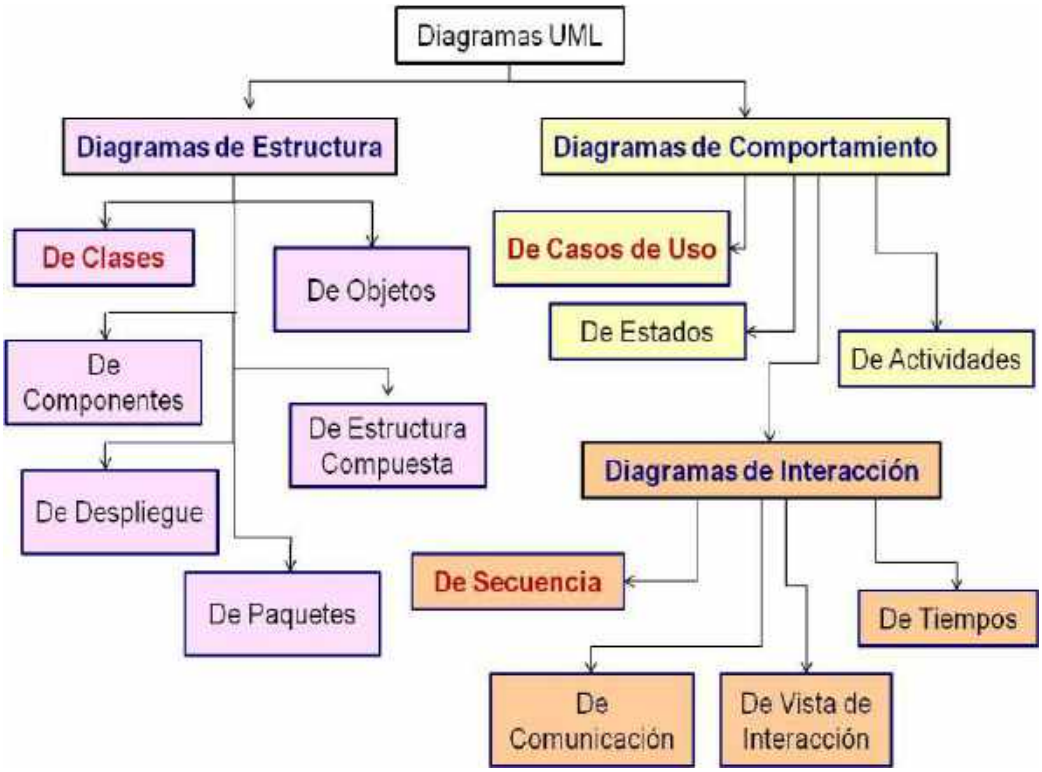


Figura 4-3: Los tipos de diagramas UML

Área	Vista	Diagramas	Conceptos principales
Estructura	Vista estática	Diagrama de clases	Clase, asociación, generalización, dependencia, realización, interfaz
	Vista de implementación	Diagrama de componentes	Componente, interfaz, dependencia, realización
	Vista de despliegue	Diagrama de despliegue	Nodo, componente, dependencia, localización
Comportamiento	Vista de casos de uso	Diagrama de casos de uso	Caso de uso, actor, asociación, extensión, inclusión, generalización de casos de uso
	Vista de máquina	Diagrama de	Estado, evento,

	de estados	estados	transición, acción
	Vista de actividad	Diagrama de actividad	Estado, actividad, transición de terminación, división, unión
	Vista de interacción	Diagrama de secuencia	Interacción, objeto. Mensaje, activación
		Diagrama de colaboración	Colaboración, interacción, rol de colaboración, mensaje
Gestión	Vista de gestión del modelo	Diagrama de componentes	Paquete, subsistema, modelo

Tabla 4-1: Tabla complementaria de clasificación de los diagramas UML

Destacamos en la Figura 4-3 los diagramas más utilizados. Para los diagramas de estructura son los *diagramas de clases*. Dentro de los diagramas de comportamiento están los *diagramas de casos de uso*. Dentro de los diagramas de comportamiento se encuentran los **diagramas de Interacción**, que hacen énfasis en *el flujo de control y de datos entre los elementos* del sistema modelado. El *diagrama de secuencia* es el más utilizado en esta subcategoría. En este capítulo estudiaremos los tres diagramas UML más utilizados: los de clases, los de casos de uso y los de secuencia.

Una descripción complementaria se proporciona en la Tabla 4-1.

4.5.1 Los diagramas de casos de uso

La idea de utilizar los casos de uso como modelo para describir los requerimientos funcionales de un sistema fue introducida por Jacobson, uno de los tres grandes contribuidores del Proceso Unificado de Desarrollo de software (UP) y UML. El modelo de casos de uso es un mecanismo que permite capturar, hacer visibles y comprensibles los objetivos y requerimientos del sistema. Los casos de uso son requerimientos funcionales que indican qué hará el sistema y quién lo hará.

Los diagramas de casos de uso describen las relaciones y las dependencias entre un grupo de *casos de uso* y los actores participantes en cada uno de los casos de uso. No están pensados para representar el diseño y no pueden describir los elementos internos de un sistema. Sirven para facilitar la comunicación con los futuros usuarios del sistema; y con el cliente, y resultan especialmente útiles para determinar las características necesarias del sistema desde el punto de vista de los usuarios. En otras palabras, los diagramas de casos de uso describen *qué* es lo que debe hacer el sistema, pero no *cómo*, por lo tanto, son muy útiles para ilustrar requerimientos.

Como se ilustra de forma genérica en la Figura 4-4, un caso de uso se define en términos de actores y escenarios.

- Un actor representa cualquier tipo de entidad externa caracterizada de un comportamiento y que interactúa con el sistema. Ejemplos de actores son: una persona identificada por un rol (operador, usuario, gerente, administrador, etc.), un departamento, una organización, otro sistema, etc.
- Un escenario es una secuencia determinada de acciones e interacciones entre los actores y el sistema en cuestión (a través de los requerimientos que caracterizan la funcionalidad de dicho sistema).

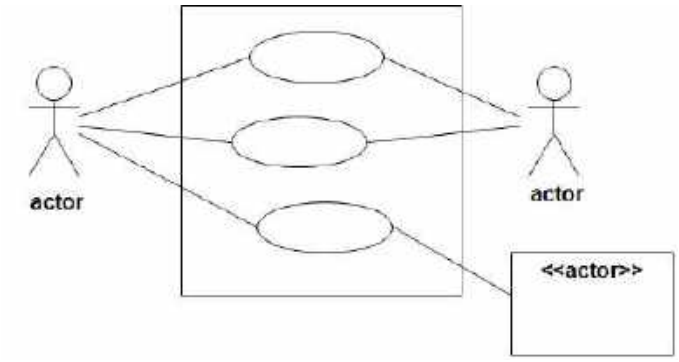


Figura 4-4: Forma genérica de un caso de uso

Una de las ventajas de estos diagramas es que se puede especificar cuáles son las funcionalidades a las que tiene acceso cada uno de los diferentes tipos de usuario de un sistema, como se ilustra en la Figura 4-5.

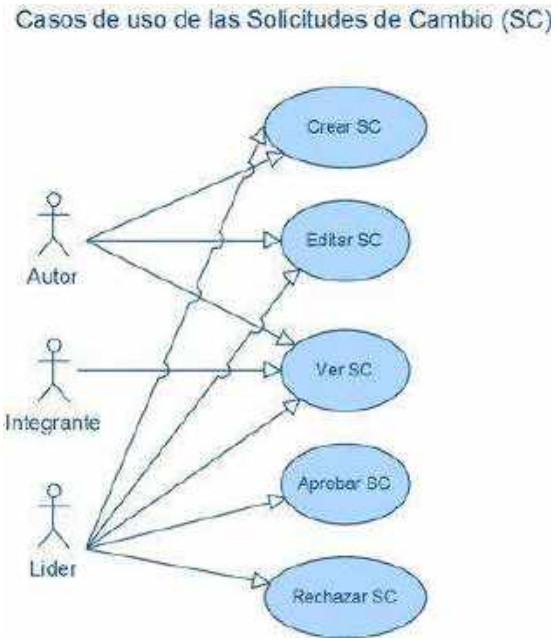


Figura 4-5: Ejemplo de diagrama de casos de uso con diferentes roles de usuario

Ejemplo.- Hacer el diagrama de casos de uso de un sistema con dos menús. Cada menú con las siguientes opciones:

1.- Libros.-
- Dar de alta un libro
- Dar de baja un libro

2.- Servicios.-
- Préstamo
- Devolución

La Figura 4-6 muestra el diagrama de casos de uso de este ejemplo.

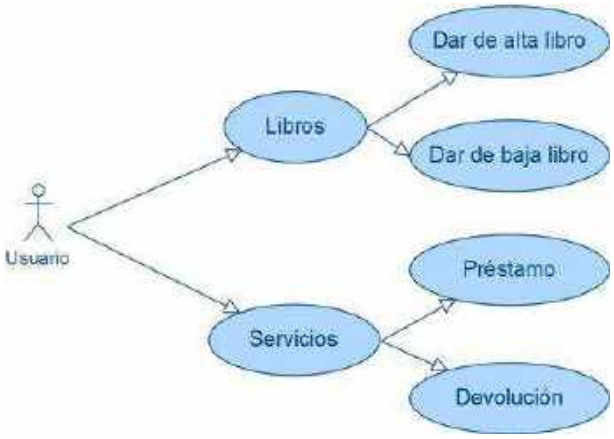


Figura 4-6: Ejemplo 2 de diagrama de casos de uso.

4.5.2 Los diagramas de clases

Un diagrama de clases es un diagrama de estructura estático que muestra las diferentes clases de objetos que componen un sistema y cómo se relacionan unas con otras. Las relaciones estáticas más utilizadas son las de asociación, composición y herencia. Los componentes principales de un diagrama de clases son las clases y todas las posibles relaciones existentes entre éstas (asociación, generalización-especialización, agregación/composición, realización y uso). La Figura 4-7 ilustra los principales tipos de relaciones que se pueden establecer entre clases.

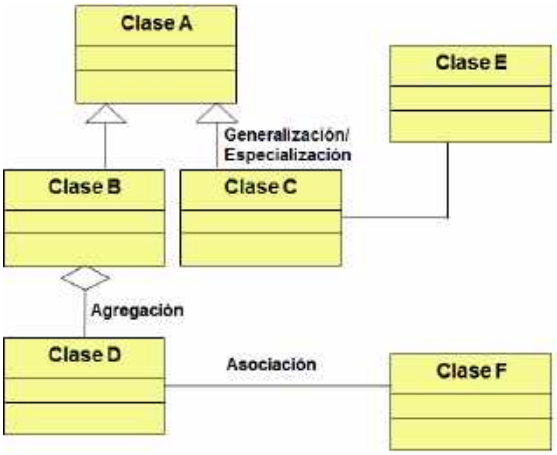


Figura 4-7: Los principales tipos de relación que se pueden establecer entre las clases

El ejemplo de la Figura 4-8, ilustra relaciones de herencia, en las que varias clases son descendientes de una clase abstracta llamada `Menu` y que, por lo tanto, heredan propiedades de ésta.

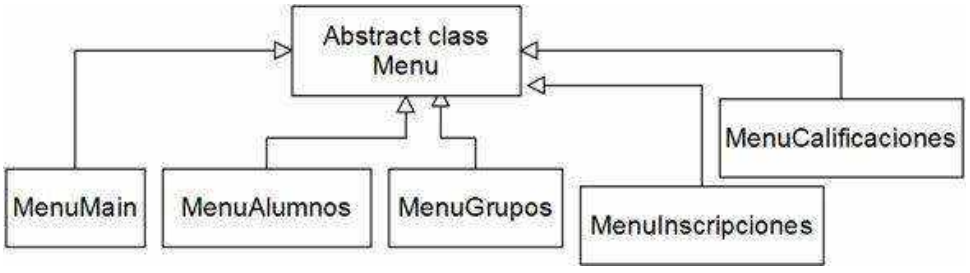


Figura 4-8: Ejemplo de diagrama de clases en el que está presente la relación de herencia

Además de las relaciones estáticas entre clases, un diagrama de clases también puede mostrar los métodos y atributos de cada una de las clases. En el ejemplo de la Figura 4-9 existe una asociación de la clase `Inscripcion` a la clase `Alumno`, y de la clase `Inscripcion` a la clase `Grupo`, y además están documentados los métodos y atributos de cada clase.

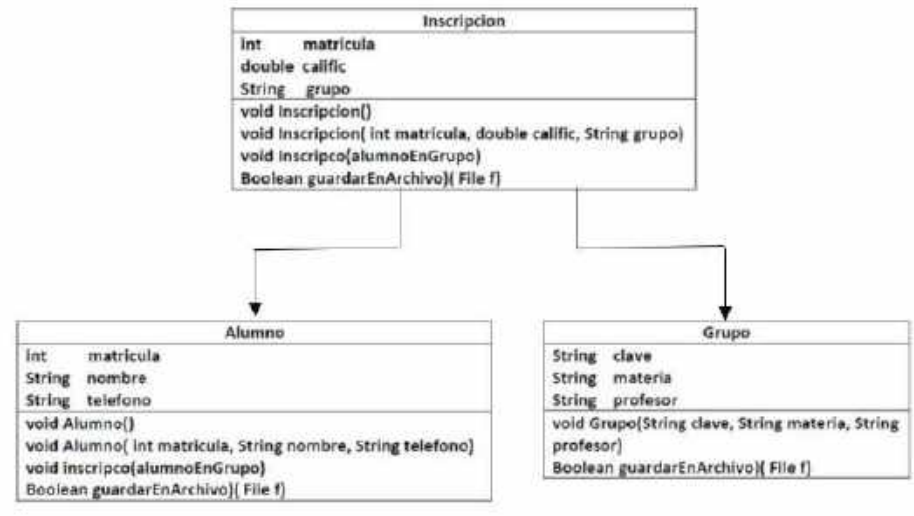


Figura 4-9: Ejemplo de diagrama de clases detallado

4.5.3 Los diagramas de interacción: diagramas de secuencia y diagramas de colaboración

Los diagramas de interacción modelan el comportamiento del sistema a través de la interacción entre los objetos que lo conforman. Describen las secuencias de paso de mensajes entre los objetos que implementan el comportamiento del sistema.

Los diagramas de secuencia sirven para expresar el orden en el que suceden las cosas. Muestran la secuencia del intercambio de mensajes entre los módulos que, en caso de la Programación Orientada a Objetos significa el momento y la forma en la que se invocan los métodos. Ponen especial énfasis en el orden y el momento en que se envían los mensajes a los objetos y bajo qué condiciones se sigue una u otra secuencia. Los componentes del diagrama de secuencia son los objetos, la línea de vida de los objetos y los mensajes. Los componentes del diagrama de colaboración son los objetos, los enlaces y los mensajes. La Figura 4-10 ilustra los componentes genéricos de un diagrama de secuencia.

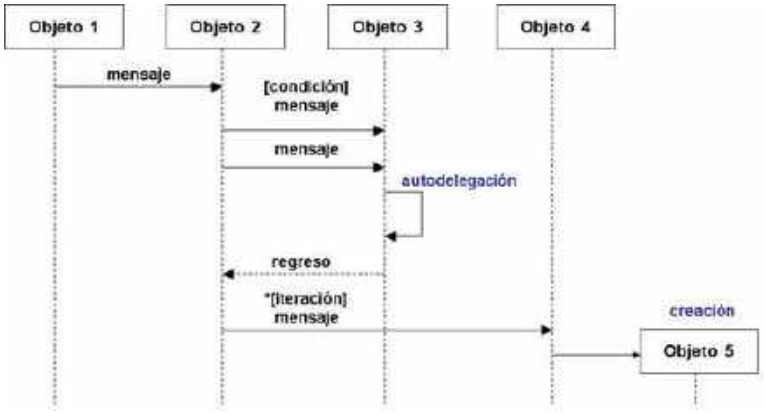


Figura 4-10: Componentes genéricos de un diagrama de secuencia

Como se ilustra en la ***Figura 4-10*** y en la Figura 4-11 los objetos o módulos se representan por líneas verticales, con su nombre en la parte más alta. El tiempo se representa en forma vertical y se incrementa hacia abajo. Los mensajes se envían de un objeto a otro en forma de flechas con el nombre del mensaje y sus parámetros. Las flechas con línea continua representan los llamados a los métodos y las flechas con línea punteada representan el regreso del llamado a un método. Se pueden insertar, como en este caso, comentarios sobre la línea vertical de cada módulo para indicar las acciones que se llevan a cabo en un momento dado dentro de un módulo. No se especifica cómo lo hace, sino solamente qué es lo que hace.

Un diagrama de colaboración representa la interacción que ocurre entre los objetos de un caso de uso, pero sin mostrar explícitamente la secuencia de la interacción en el tiempo. A diferencia de un diagrama de secuencia, el cual se define en términos de tres símbolos básicos (objeto, línea de vida y mensaje), el diagrama de colaboración no incluye las líneas de vida y define un nuevo símbolo: los enlaces entre objetos, representados por líneas (como se representan las asociaciones en un diagrama de clases). Otra diferencia entre el diagrama de secuencias y el diagrama de colaboración, consiste en que este último enumera los mensajes que se pasan entre los objetos.La Figura 4-12 muestra un ejemplo de un diagrama de colaboración.

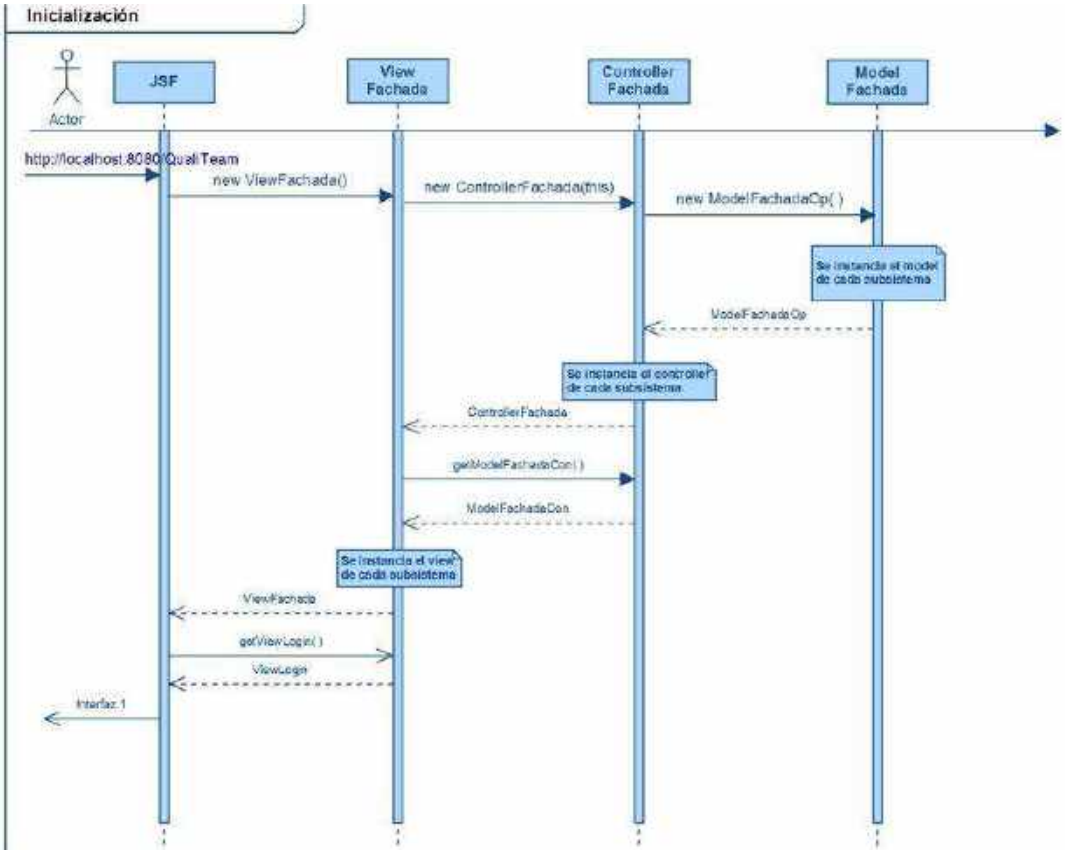


Figura 4-11: Ejemplo de diagrama de secuencia

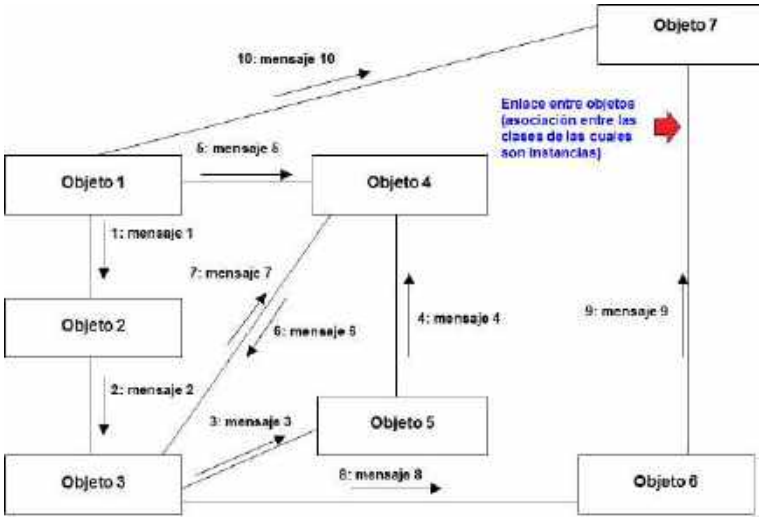


Figura 4-12: Ejemplo de diagrama de colaboración

4.5.4 Paquetes de clases

Un paquete de clases es una forma de organizar un grupo de clases relacionadas. Un paquete de clases funge como un contenedor de clases. Las clases en un paquete pueden estar relacionadas a partir de diferentes criterios: aspecto del dominio del problema que modelan, tipo de servicio que proporcionan, tipo de capa lógica donde se ubican, etc.

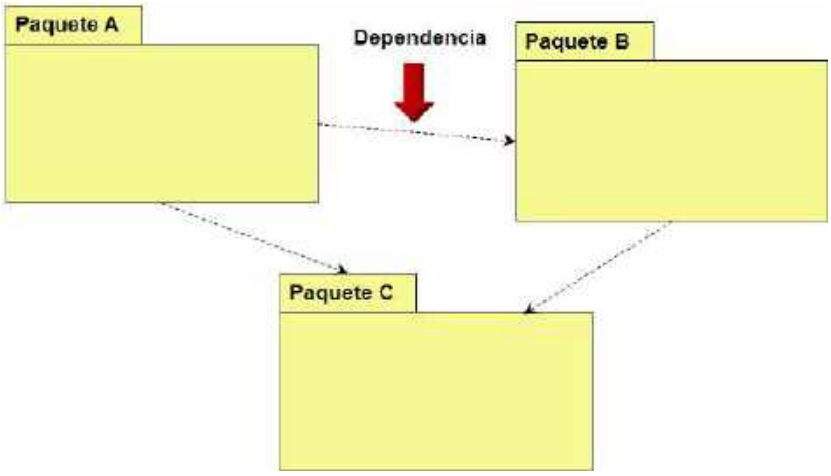


Figura 4-13. Diagrama de paquetes.

Un paquete (*package* en Java) representa un conjunto de clases correlacionadas. Es posible definir una clase como perteneciente a un cierto *package*. Así como existe una estrecha correspondencia entre clase y file, también existe una estrecha correspondencia entre *package* y *directorio*. Por lo tanto, en general un *package* de nombre X es representado por un directorio de nombre X, donde X es una cadena con todos los caracteres en minúscula (por convención).

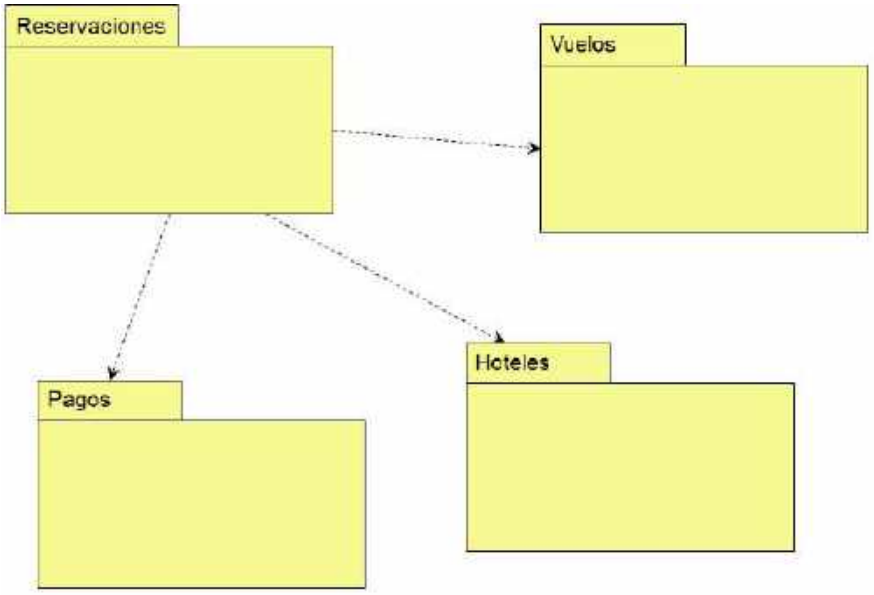


Figura 4-14. Diagrama de paquetes propuesto durante el diseño de un sistema Web de reservaciones y ventas de vuelos y hotel.

Solo las clases o interfaces públicas de un *packag*e son accesibles desde el exterior del *package* en el cual vienen definidas. Existen varias formas de acceder a un miembro público (clase o interfaz) de un *package* desde fuera de éste:

- Referirse al miembro del *package* por su nombre completo (P.C).
- Importar en el código el miembro del *package.*
- Importar el *package* completo en el código.

Los diagramas de paquetes resultan de gran utilidad para concebir la estructura global de un sistema en término de los módulos que la integran. Un diagrama de paquetes exhibe las dependencias entre los paquetes de clases que componen el modelo de diseño. El paquete "A" establece una relación de dependencia con el paquete "B", si al menos un elemento del paquete "A" requiere de al menos un elemento del paquete "B". Estos elementos son comúnmente clases. Como se puede apreciar en la Figura 4-13, los elementos de un diagrama de paquetes son principalmente los paquetes y las relaciones de dependencia entre los mismos. Por otra parte, la Figura 4-14, muestra el diagrama de paquetes correspondiente a un sistema Web de reservaciones y ventas de vuelos y hotel.

4.6 La adopción de UML en la ingeniería de software

UML es actualmente el lenguaje de modelado más utilizado. Es muy útil para ilustrar los requerimientos y el diseño de un sistema de software. Sin embargo, sus inconvenientes y defectos han sido históricamente identificados y señalados [Budgen et. al, 2011]. Existe un estudio [Grossman et. al, 2005] dedicado a investigar la adopción y el uso de UML en la comunidad de desarrollo de software. Uno de los resultados de este estudio es que existe un "cierto grado de confusión entre la comunidad de usuarios de UML". En este estudio se mencionan los siguientes problemas como las posibles causas que hacen de UML un estándar inmaduro: su naturaleza evasiva, su complejidad, es poco comprendido y, a veces, se usa de manera inconsistente. A pesar de todos estos problemas, UML se ha convertido en el estándar para el desarrollo de sistemas, por lo que es importante continuar haciendo cambios para mejorarlo.

4.7 Los Diagramas de Transición entre Interfaces de Usuario (DTIU)

Los Diagramas de Transición entre Interfaces de Usuario (DTIU) [Gómez y Cervantes, 2013], son una herramienta que sirve para modelar el flujo entre las diferentes interfaces que se le presentan al usuario en un sistema de software.

Los DTIU son una herramienta que permite modelar sistemas de software de forma clara e intuitiva para que los clientes (normalmente sin capacitación en lenguajes de modelado) puedan comprender los diagramas y participen en el proceso de extracción de los requerimientos. Los DTIU facilitan la transición de la fase de requerimientos a la de diseño y una de sus grandes ventajas es que no se presta a confusiones ni a ambigüedades.

En la Figura 4-15 se ilustra un ejemplo de entrada a un sistema (por medio de cuenta de usuario y contraseña) modelado con un DTIU. En este ejemplo, introducir UserId y Password correctamente o incorrectamente son dos acciones distintas, y cada una tiene su correspondiente transición. El sistema envía un mensaje de error y regresa a la interfaz original cuando el usuario proporciona UserId y/o Password incorrecto mientras que en el otro caso avanza hacia la interfaz principal del sistema.

En este diagrama se puede apreciar claramente cuáles son las operaciones que el usuario puede llevar a cabo en cada una de las interfaces que están en **negrita**, las interfaces que no se destacan en negrita, como el "Home del sistema" en nuestro ejemplo, indican que no se han especificado en el diagrama todas sus transiciones, y que éstas se ilustran por separado en otro DTIU.

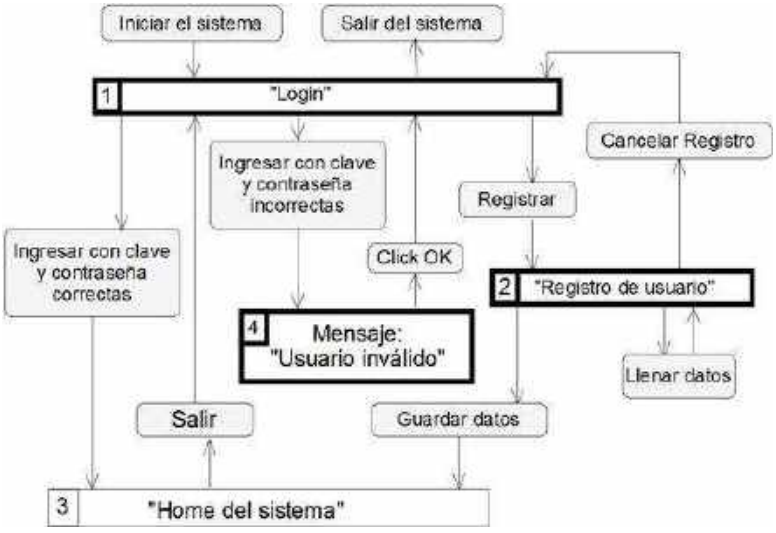


Figura 4-15: Ejemplo de Diagrama de Transición entre Interfaces de Usuario (DTIU)

Con los DTIU también es posible representar las acciones que puede llevar a cabo cada uno de los diferentes tipos de usuario de un sistema. Por ejemplo, en la Figura 4-16 existen dos tipos de usuario: A y B. Mientras que el usuario A solo puede entrar al menú del juego 1, el usuario B puede además editar, crear o borrar juegos de cartas.

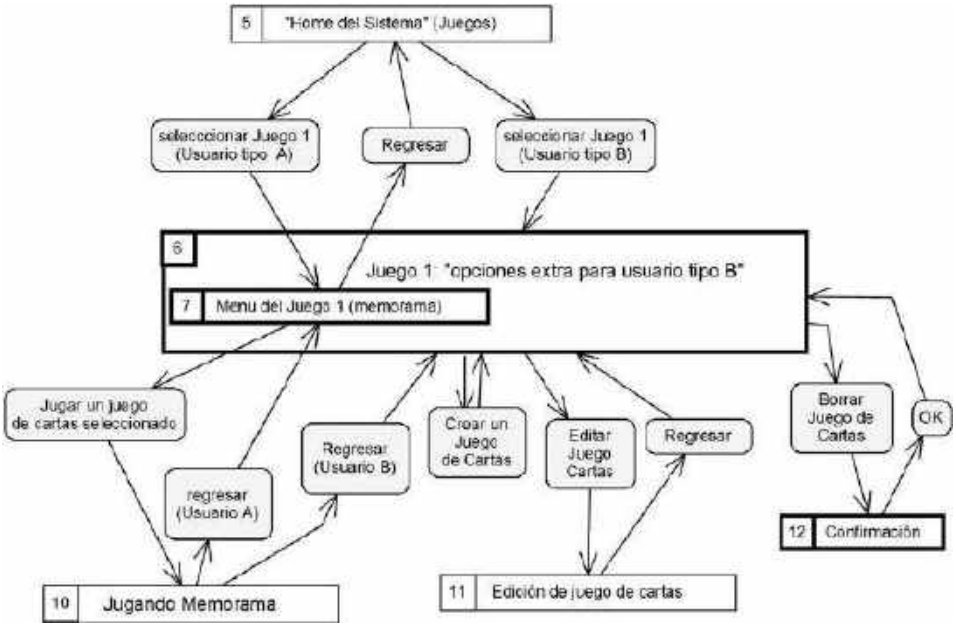


Figura 4-16 DTIU con diferentes roles de usuario

El usuario de tipo A accede a la interfaz 7 desde la que puede ir a jugar a la interfaz 10 o regresar a la 5. El usuario tipo B accede a la interfaz 6 desde la cual puede hacer todas las transiciones que salen de la interfaz 7 más aparte las que salen de la interfaz 6.

4.7.1 Ejercicio de comprensión de un DTIU

4.7.1.1 Preguntas

En la Figura 4-17 se presenta el DTIU de un sistema para organizar un congreso. Contestar las preguntas que se formulan después del diagrama.

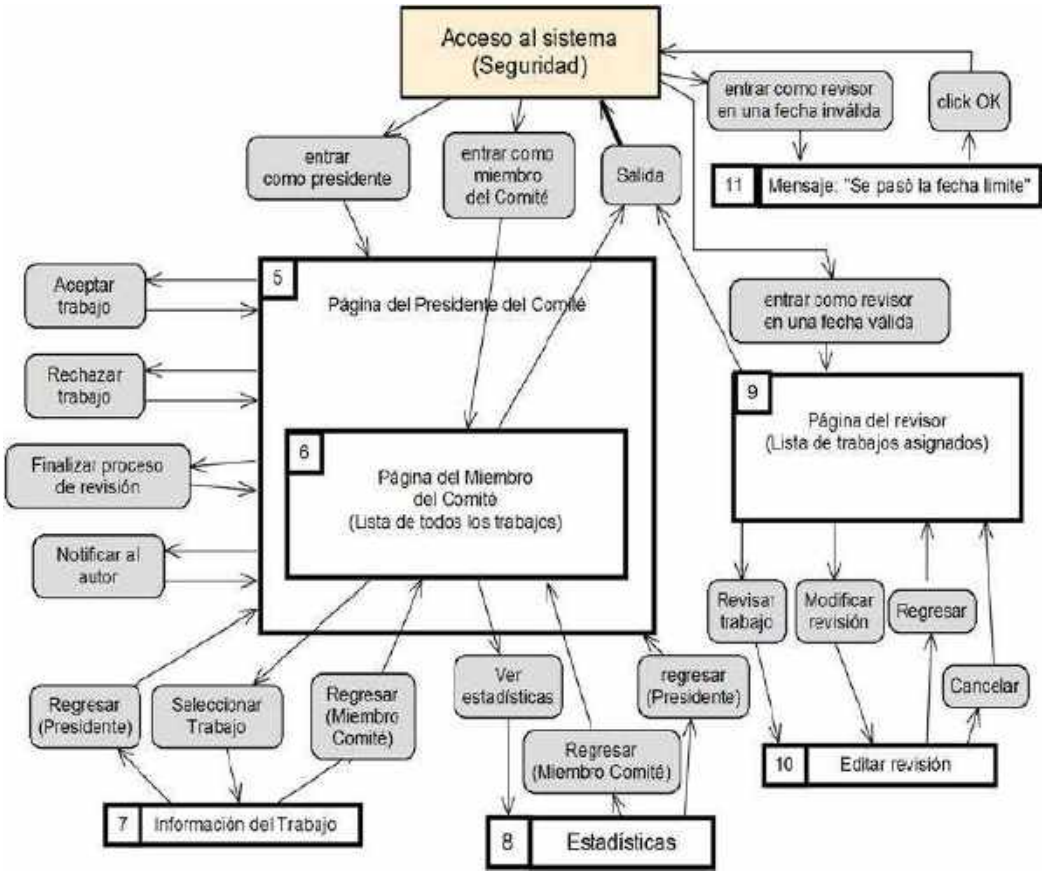


Figura 4-17: DTIU de un sistema para organizar un congreso

1.- ¿Puede el presidente del congreso aceptar o rechazar un trabajo?

☐ Sí ☐ No

2.- ¿Puede cualquier miembro del comité aceptar o rechazar un trabajo?

☐ Sí ☐ No

3.- ¿Quién notifica al autor sobre la decisión final para su trabajo?

☐ El presidente ☐ Un miembro del comité ☐ Un revisor

4.- ¿Qué sucede cuando un revisor intenta hacer una revisión después de la fecha límite?

5.- Selecciona a los usuarios que pueden ver las estadísticas (puedes elegir una o varias opciones)

☐ El presidente ☐ Un miembro del comité ☐ Un revisor

6.- ¿Puede el presidente del congreso hacer la revisión de un trabajo?

☐ Sí ☐ No

7.- ¿Cuál es el nombre de la interfaz que se le presenta al usuario después de que eligió "Seleccionar trabajo"?

8.- ¿Es necesario que el usuario pase por el sub-sistema de seguridad para poder acceder a las páginas del sistema?

☐ Sí ☐ No

4.7.1.2 Respuestas del ejercicio

1.- Si

2.- No

3.- El presidente

4.- Se presenta el mensaje "Se pasó la fecha límite

5.- El presidente, un miembro del comité

6.- No

7.- "Información del Trabajo"

8.- Si

4.8 Herramientas para modelar

Hay una gran cantidad de herramientas software para hacer diagramas (modelos), algunos de paga y otros gratuitos. En la Figura 4-18 lustramos algunos ejemplos de herramientas gratuitas, como Dia, ArgoUML, Umbrello, BoUML y Poseidón. En la Figura 4-19 se ilustra UMLPaceStar (de paga). Esta herramienta contiene una gran variedad de tipos de diagramas, su uso es sencillo y la presentación que se logra en los diagramas es muy buena.

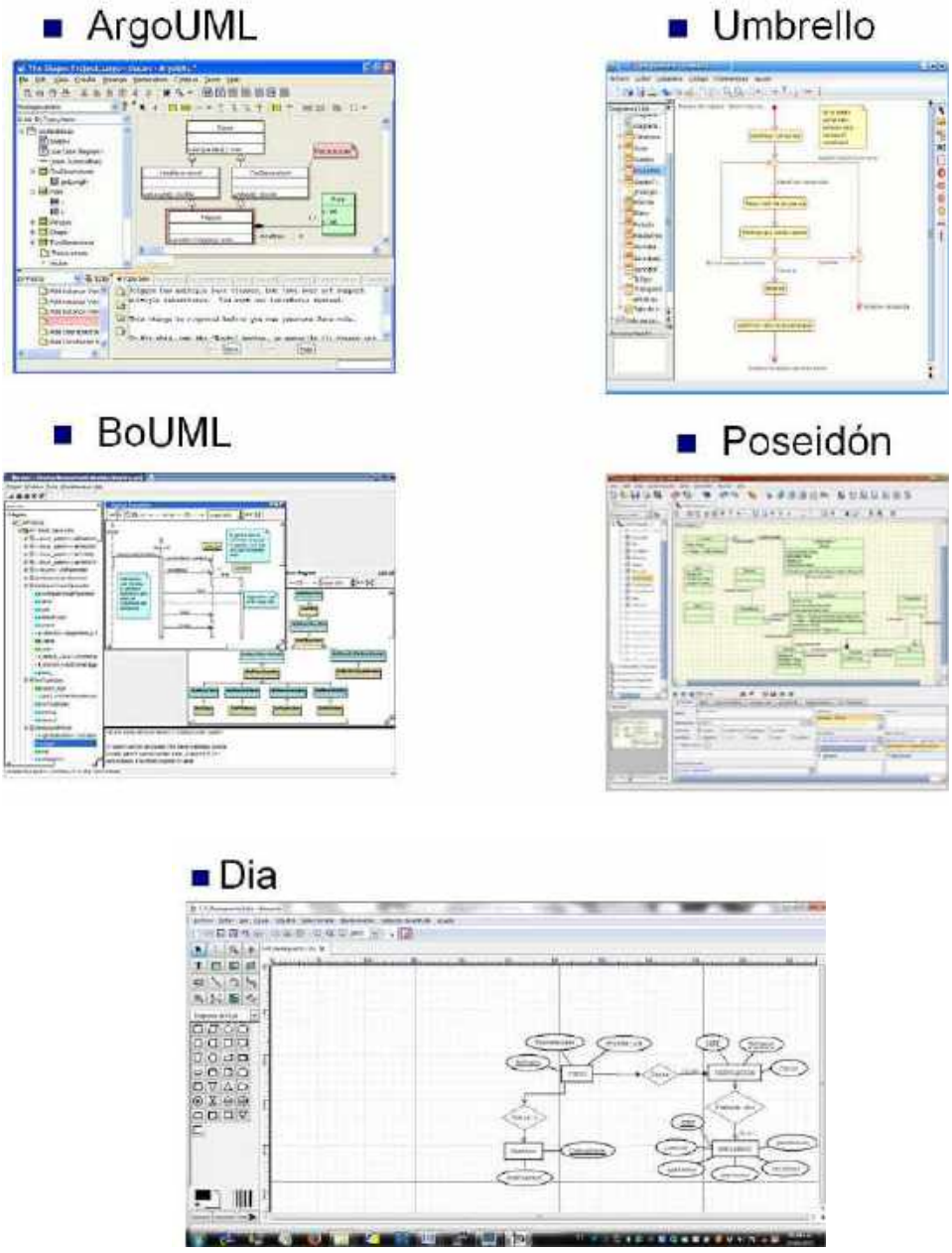


Figura 4-18: Ejemplos de herramientas para modelar (gratuitos)

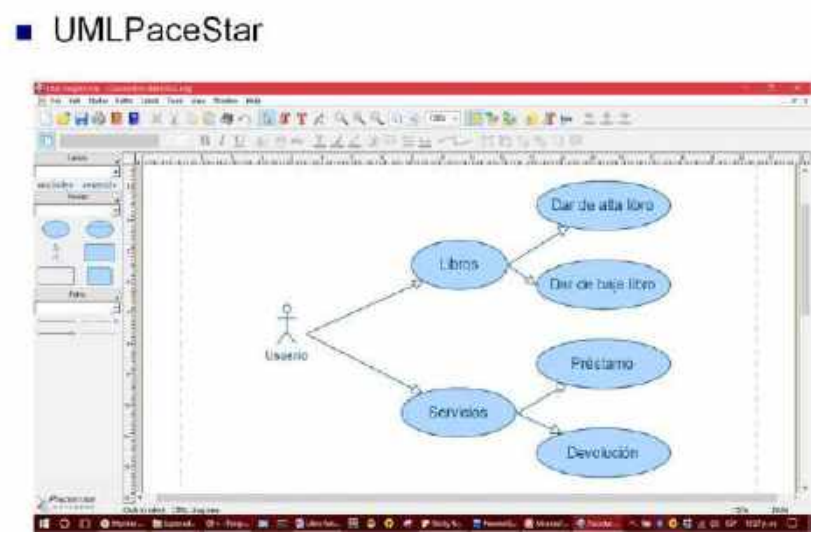


Figura 4-19: Ejemplo de herramientas para modelar (de paga)

4.9 Resumen

Un modelo es una *representación* de un concepto, de un objeto o de un sistema, que permite explicar algunas propiedades específicas, como, por ejemplo: la forma, el comportamiento, la estructura o la relación de una entidad con otras entidades.

UML (Unified Modelling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. UML se usa como lenguaje gráfico para visualizar, especificar, construir y documentar sistemas. UML es visual, y mediante su sintaxis se modelan distintos aspectos que permiten una mejor interpretación y entendimiento de un problema o de un sistema computacional.

UML tiene diagramas para modelar la estructura, también llamada "vista estática" de un sistema, y diagramas para modelar el comportamiento. El diagrama de estructura más utilizado es el diagrama de clases. Y los diagramas de comportamiento más utilizados son los diagramas de casos de uso y los diagramas de secuencia.

Los Diagramas de Transición entre Interfaces de Usuario (DTIU), son una herramienta que sirve para modelar el flujo entre las diferentes interfaces que se le presentan al usuario en un sistema de software. Los DTIU permiten modelar sistemas de software de forma clara e intuitiva para que los clientes puedan comprender los diagramas y participen en el proceso de extracción de los requerimientos. Los DTIU facilitan la transición de la fase de requerimientos a la de diseño y una de sus grandes ventajas es que no se presta a confusiones ni a ambigüedades.

4.10 Cuestionario

1.- ¿Qué es un modelo?

2.- ¿Para qué sirve UML?

3.- Explica cuáles son los aspectos que modelan cada uno de los siguientes grupos de diagramas UML:

- **Diagramas de Estructura.-**

- **Diagramas de Comportamiento.-**

- **Diagramas de Interacción.-**

4.- Explica en qué consiste un diagrama de clases.

5.- Dibuja el diagrama de Casos de Uso de un sistema administrador de un consultorio dental en el que hay dos menús, cada uno con las siguientes opciones:

1.- Pacientes.- Con las siguientes opciones:

- Dar de alta un paciente
- Consultar pacientes

2.- Citas.- Con las siguientes opciones:

- Alta de un mes de trabajo
- Consultar citas del mes
- Agendar una cita
- Cancelar una cita

6.- Explica en qué consiste el Diagrama de Transición entre Interfaces de Usuario (DTIU).

5 Diseño del software

5.1 Objetivos específicos del capítulo

- Conocer cuáles son las características de un buen diseño de software.
- Comprender la diferencia entre diseño de alto nivel y diseño detallado.
- Comprender la diferencia entre diseño estructurado y diseño orientado a objetos.

5.2 Introducción

El diseño es un proceso en el que se determina la estructura del sistema de software y de sus datos antes de iniciar su codificación. Además, proporciona en algunos casos la base de la lógica para codificar de tal manera que se cumpla con la especificación de requerimientos. Durante la etapa de diseño se produce el *Documento de Diseño.*

Durante el análisis de requerimientos se responde a las preguntas ¿QUÉ? y ¿CUÁL?, mientras que en el diseño del sistema software se responde a la pregunta ¿CÓMO? Durante el diseño, se toman decisiones acerca de la forma en que se resolverá el problema, primero desde un nivel elevado de abstracción y después empleando cada vez niveles de refinamiento que muestren más detalles. La fase de diseño del sistema software parte de los resultados del análisis de los requerimientos, los cuales deben proporcionar los elementos claves para determinar la estructura del software. Durante el diseño del sistema software se decide la arquitectura del mismo, el diseño global, el diseño de los componentes, el diseño de las interfaces de los componentes y el diseño de los datos.

Se desea que un sistema de software sea **confiable**, es decir, con baja probabilidad de fallas, que sea **robusto**, es decir, que se comporte razonablemente en circunstancias que no fueron anticipadas en los requerimientos y que sea **eficiente**, o sea, que haga buen uso de los recursos.

Por otra parte, se considera que un diseño es bueno cuando es simple y comprensible, cuando es flexible, es decir que se pueda adaptar a otras circunstancias agregando o modificando módulos y de preferencia que se pueda reutilizar, es decir, que se puede utilizar en otros sistemas. Braude (2003) propone las siguientes metas de diseño:

Diseñar para el cambio.- Para que el diseño admita con facilidad futuras alteraciones en los requerimientos del sistema.

Diseñar para la extensión.- Para que el diseño facilite la adición de nuevas funcionalidades.

Diseñar para la eficiencia.- Se busca satisfacer algún objetivo particular: alta velocidad, tamaño pequeño, etc.

Diseñar para la sencillez.- Lograr un diseño fácil de comprender y de implementar.

Existe un compromiso entre la sencillez y las demás metas del diseño pues, al diseñar para lograr cualquiera de estas metas, se disminuye la sencillez.

5.3 Diseño de alto nivel y diseño detallado

El proceso de diseño se divide en dos fases: el ***diseño de alto nivel***, también llamado diseño arquitectónico y el ***diseño detallado*.** El diseño de alto nivel (en inglés TLD: Top Level Design), define los módulos que componen al sistema y cómo éstos interactúan entre sí. Mientras que el diseño detallado (en inglés DD: Detailed Design) define los datos del sistema sin llegar a los detalles de codificación. Los datos del sistema son los que se guardan en archivos o en bases de datos y los que se envían entre los módulos.

5.3.1 Diseño de alto nivel

Durante el diseño de alto nivel se determina una *solución arquitectónica*, es decir, una descomposición del sistema en subsistemas. Los subsistemas son *módulos que pueden descomponerse en sub-módulos*. A su vez, cada sub-módulo se descompone en sub-módulos a este proceso se le conoce como *modularización*. Además, se lleva a cabo el *ensamblaje*, que es la definición de la comunicación y dependencia entre los módulos. También se toma en cuenta la posibilidad de aplicar una arquitectura ya existente reutilizando algún patrón de diseño conocido. Además, se considera en qué ambiente va a funcionar el sistema. El diseño de alto nivel (diseño arquitectónico) según la IEEE-1471 [Maier et. al, 2004] es "la organización fundamental de los componentes de un sistema, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución".

El diseño de alto nivel o diseño arquitectónico tiene como principal tarea especificar la organización del software, considerando los componentes que integrarán el sistema, las interfaces y comportamientos que caracterizan a estos componentes, así como la forma en que se establece la comunicación, interacción y colaboración entre estos componentes. La arquitectura del software resultante de este proceso de diseño a alto nivel deberá integrar todos los aspectos del software: lógicos, de proceso, de componentes, físicos, etc.

El diseño de alto nivel se ocupa de tres puntos esenciales: 1).- La definición de la arquitectura (módulos que componen el sistema), 2).- La definición de la estructura de los datos que van a usar los módulos, 3).- La definición de las interfaces con el usuario y entre los módulos, y 4).- La propuesta de una solución que cumpla con los requerimientos no funcionales, tales como fiabilidad, seguridad, eficiencia en tiempo/espacio, etc.

5.3.2 Diseño detallado

El diseño detallado describe a detalle cada uno de los datos que comparten los módulos de la solución, sirve como guía para codificar de tal forma que se satisfagan los requerimientos, y también sirve como explicación de lo que debería estar codificado, lo que facilita las pruebas y el mantenimiento futuro.

Como se ilustra en la Figura 5-1, los módulos que componen al sistema y sus interfaces son producto del diseño de alto nivel. El producto del diseño de alto nivel es la entrada para comenzar el diseño detallado. Durante el diseño detallado se especifican los datos y la lógica de lo que se tiene que hacer y para lograrlo se utilizan diagramas de secuencia, de flujo, de estado, de clases, etc. Podemos decir que el diseño detallado especifica los requerimientos de cada módulo. El diseño detallado sirve como entrada para el proceso de codificación.

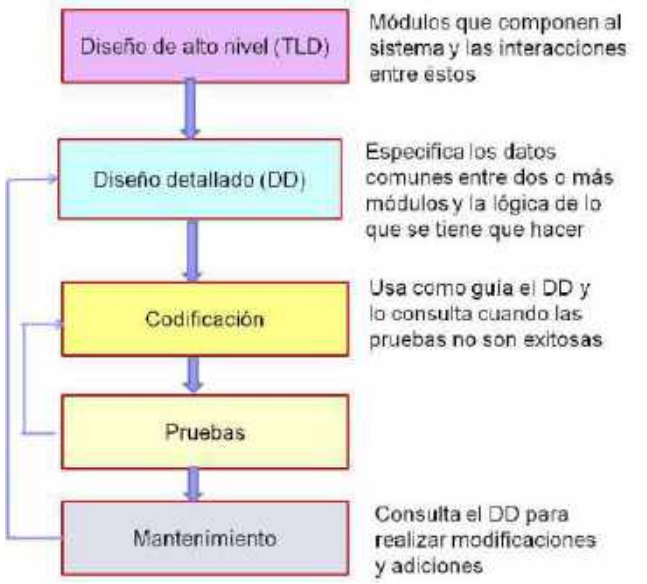


Figura 5-1: Relación entre el diseño y las otras etapas de desarrollo de software

En el documento de diseño normalmente se incluyen diagramas que describen el diseño lógico como, por ejemplo, los diagramas de clases, de interacción y de estados. El diseño detallado parte del modelo de diseño arquitectónico, y modela en detalle cada una de las partes o componentes (módulos, clases, funciones, procedimientos, etc.) que integran la arquitectura propuesta, aplicando un diseño modular efectivo. Como su nombre lo indica, en el diseño detallado se diseña de forma detallada (precisa, minuciosa), a través de los modelos y diagramas más apropiados, la comunicación y dependencia entre los módulos diseñados.

Durante el diseño detallado se debe verificar que cada uno de los componentes diseñados a detalle satisfaga los requerimientos funcionales especificados en la etapa de análisis, y, por lo tanto, las necesidades y expectativas del cliente. El modelo de diseño debe proporcionar una especificación de diseño a un nivel de detalle tal, que el proyecto quede listo para la fase de implementación/codificación.

Es importante señalar que existe una frontera difusa entre el diseño detallado y la codificación. Como señala Sommerville (2007), "el diseño de la estructura de datos y de los algoritmos se puede retrasar hasta la etapa de implementación". La decisión del nivel de detalle (la profundidad) en el diseño detallado estará en función del tamaño del proyecto y del nivel de calificación de los programadores. Si se cuenta con personal muy calificado para codificar, entonces se podrán delegar más responsabilidades a los programadores. Sin embargo, cuando el proyecto es muy grande o los programadores son novatos, la descripción detallada de los datos y de los algoritmos en el diseño detallado disminuirá el riesgo de cometer errores durante la codificación.

Los puntos esenciales del Diseño Detallado son:

1) Diseño detallado de datos
   a) Entradas
   b) Salidas
   c) Base de datos
2) Diseño detallado de las interfaces
   a) Del sistema con el usuario
   b) Entre los componentes del sistema

3) Diseño detallado de los módulos
   a) Definición de secuencias y condiciones (con diagramas de secuencia, de flujo, de estados,...)
   b) Definición de clases y las relaciones entre éstas (para POO)

La Figura 5-2 ilustra los roles de las personas que intervienen en el desarrollo del sistema de software, desde la extracción de requerimientos hasta la codificación. Como puede observarse, el trabajo del programador depende directamente de lo que se produce durante el diseño.

Figura 5-2: Roles desde los requerimientos hasta la codificación (imagen disponible en internet[4])

5.4 Diseño Estructurado y Diseño Orientado a Objetos

En la sección anterior estudiamos el proceso de diseño en función de su nivel de detalle. En función del enfoque, existe el *diseño estructurado* y el *diseño orientado a objetos.* El *diseño estructurado* toma en cuenta que el código consta de bloques lógicamente estructurados en forma de: instrucciones condicionales, ciclos y bloques lógicos. Mientras que el diseño orientado a objetos, supone que el código consta de objetos ("mini-programas") que interactúan entre sí.

5.4.1 El Diseño Estructurado

El *diseño estructurado* es útil para resolver problemas de naturaleza algorítmica, es decir, dada cierta entrada se produce cierta salida. La información fluye a través de las estructuras y de llamados a funciones. La modularización se lleva a cabo por medio de la *descomposición jerárquica* del problema (ver sección 5.5). El diseño estructurado se recomienda para sistemas pequeños.

Cuando la construcción del software es guiado por el diseño estructurado, entonces:

- El sistema software es una jerarquía de módulos, con un módulo principal (también llamado programa principal) con una función de controlador.

[4] Imagen de https://www.cs.northwestern.edu/academics/courses/394/slides/reveal.js-master/images/voting.png

- El módulo principal transfiere el control a los módulos inmediatamente subordinados (o subprogramas), de modo que éstos puedan ejecutar sus funciones. Una vez que el módulo subordinado haya completado su tarea, devolverá nuevamente el control al módulo controlador.

- La descomposición de un módulo en submódulos continúa hasta que se llegue a un punto en que el módulo resultante tenga solo una tarea específica que ejecutar (lectura, salida de resultados, procesamiento de datos o control de otros módulos).

En la Figura 5-3 se ilustra la estructura genérica de un sistema de software construido siguiendo un diseño estructurado.

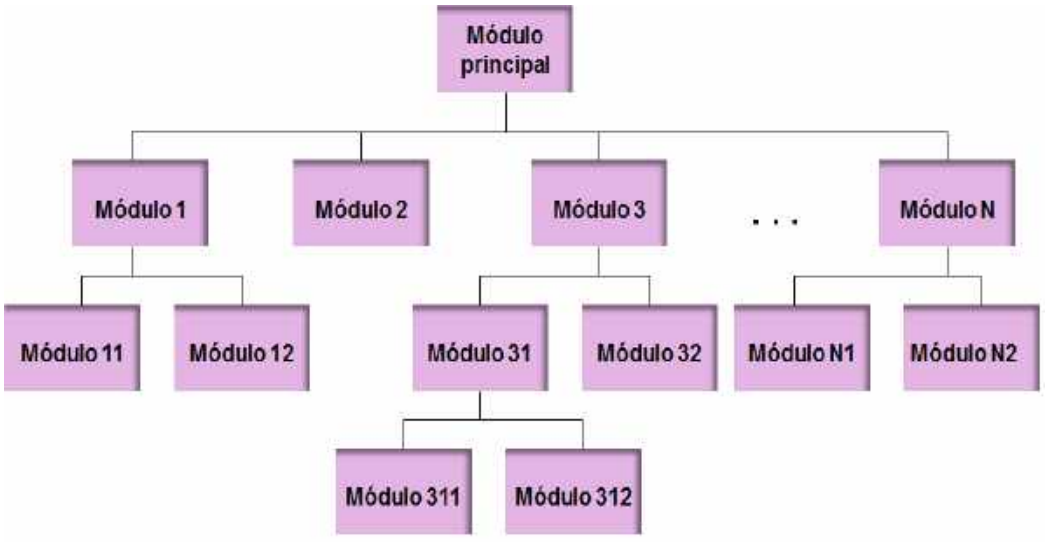


Figura 5-3: Estructura genérica de un sistema de software

Las estructuras de datos y las funciones se representan por medio de diagramas de entidad-relación, de flujo de datos y de transición de estados.

5.4.1.1 Diagrama de Flujo de Datos (DFD)

Es un diagrama en forma de red que representa el flujo de datos y las transformaciones que se aplican sobre ellos al moverse desde la entrada hasta la salida del sistema. En la Figura 5-4 se ilustran los elementos de un DFD.

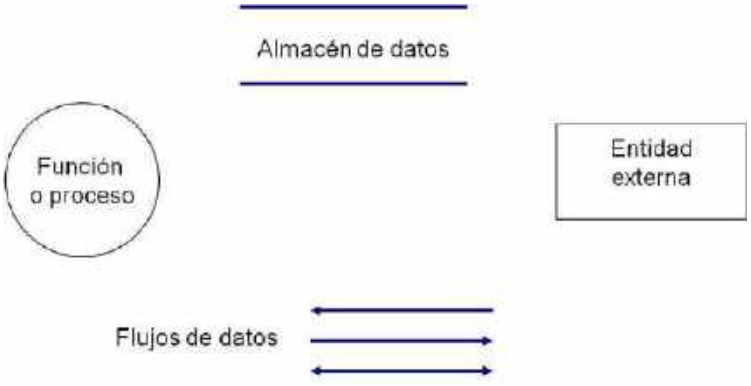


Figura 5-4: Elementos del Diagrama de Flujo de Datos (DFD)

El DFD se apoya en el Diccionario de Datos para documentar más ampliamente sus elementos, el Diccionario de Datos es un documento aparte en el que se explica en qué consisten los términos que se usan un diagrama. Un DFD es independiente del tamaño y de la complejidad del sistema, porque se puede organizar por niveles de abstracción, es decir, se comienza con un diagrama que describe el flujo de datos de manera general, y después se van haciendo diagramas adicionales para detallar los procesos que sean necesarios. En la Figura 5-5 se ilustra un ejemplo de Diagrama de Flujo de Datos en el cual se describen las actividades que tienen que ver con los clientes, productos y proveedores de una empresa.

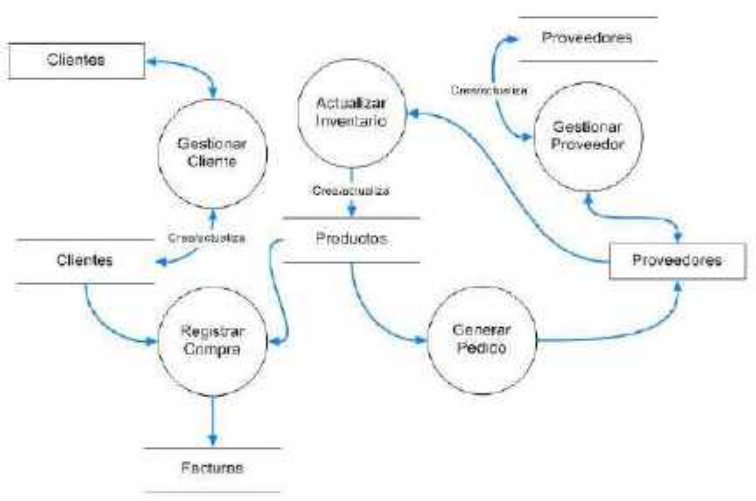


Figura 5-5: Ejemplo de un Diagrama de Flujo de Datos (DFD)

5.4.1.2 Diagrama de Transición de Estados (DTE)

El Diagrama de Transición de Estados (DTE) se utiliza cuando el comportamiento del sistema puede ser distinto para un mismo estímulo. Esto se debe a que el sistema puede estar en diferentes estados y el comportamiento depende no solamente de estímulo que éste recibe, sino también del estado en el que se encuentre. En un DTE se indican todos los estados del sistema, aquellos eventos que estén relacionados con cambios de estado y los cambios de estado que se producen. Los componentes de un DTE son los estados (nodos) y las transiciones (flechas que representan cambios de estado). Además, se pueden agregar condiciones y las acciones asociadas a las transiciones.

Un DTE relaciona eventos y estados. Cuando se recibe un evento, el estado siguiente depende del actual, así como del evento recibido. Un cambio de estado causado por un evento es lo que se conoce como transición.

Figura 5-6: Ejemplo de un Diagrama de Transición de Estados (DTE)

La Figura 5-6 ilustra un ejemplo de Diagrama de Transición de Estados de una tarjeta bancaria. Primero se crea y se encuentra en estado "Tarjeta Creada", al activarla pasa al estado "Tarjeta Activa", mientras esté en este estado se pueden realizar abonos y cargos, el diagrama termina cuando se cancela la tarjeta. En el paradigma orientado a objetos (ver siguiente sección) el DTE describe todos los estados posibles por los que puede transitar un objeto particular, como resultado de los eventos que llegan a éste.

5.4.1.3 Diagrama Entidad-Relación (DER)

Un diagrama Entidad-Relación (DER) modela el mundo real como un conjunto de objetos básicos llamados entidades y las relaciones existentes entre ellas. En la Figura 5-7 se ilustran los elementos de un DER.

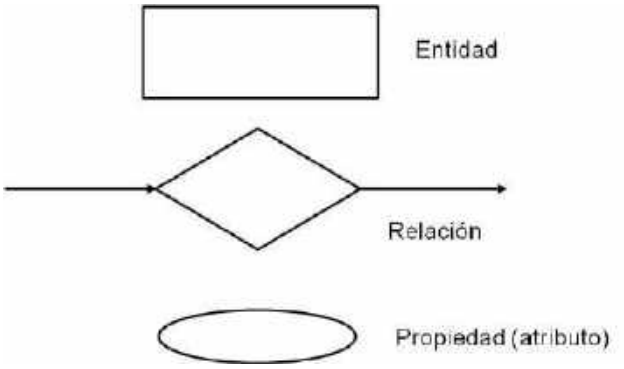


Figura 5-7: Elementos del Diagrama Entidad-Relación (DER)

El ejemplo de la Figura 5-8 modela con un DER las entidades y las relaciones que existen en una empresa en la que hay empleados, departamentos, jefes de departamento y proyectos que se llevan a cabo en los departamentos.

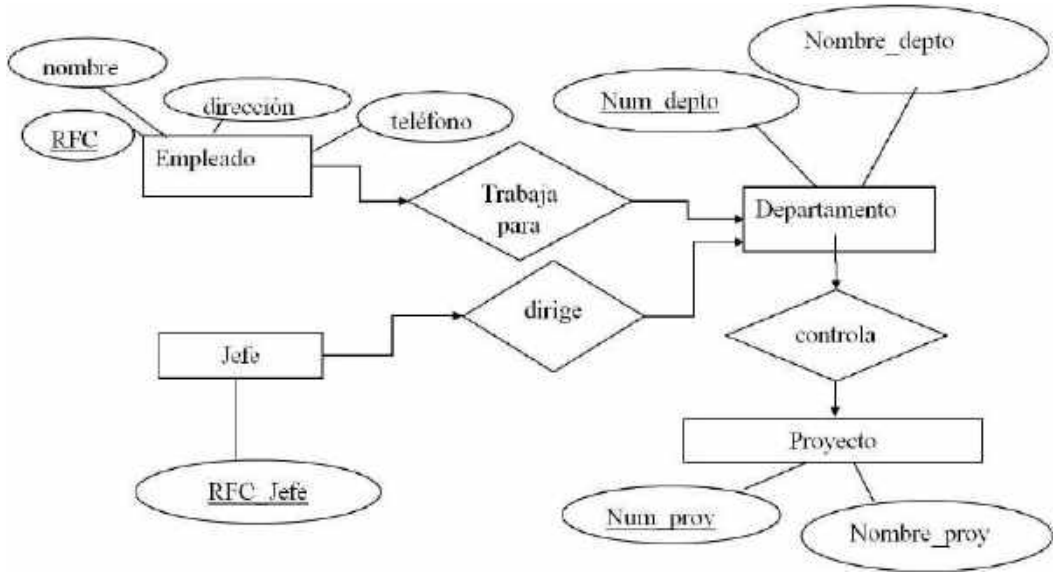


Figura 5-8: Ejemplo de Diagrama Entidad-Relación (DER)

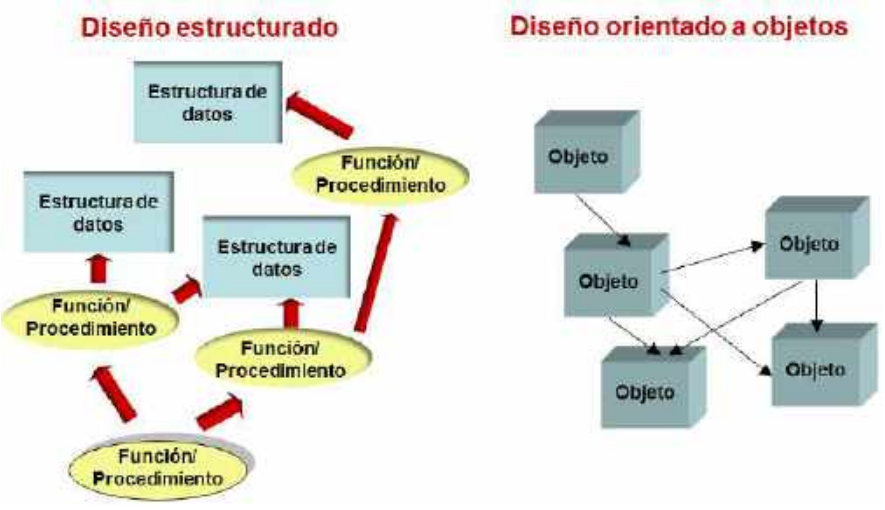
5.4.2 El Diseño Orientado a Objetos

El Diseño Orientado a Objetos se basa en objetos que tienen un estado y un comportamiento tomando en cuenta que existen interacciones entre los objetos. Este tipo de diseño es útil cuando el sistema se puede modelar de forma casi análoga a la realidad, porque así se simplifica el diseño de alto nivel. Con el Diseño Orientado a Objetos se promueve la reutilización, ya que las similitudes entre objetos se programan en forma abstracta y el programador concentra su esfuerzo en las diferencias concretas.

Bajo el principio del *encapsulamiento*, el software queda organizado y protegido, así un programador puede entender mejor el código de los otros y hay menor riesgo de que sus cambios afecten el trabajo de los demás. Por esto, el diseño orientado a objetos se usa en el desarrollo de software a gran escala, pues los equipos de programadores trabajan sobre objetos diferentes y posteriormente se integra el trabajo de todos haciendo uso de las *interfaces* entre los objetos.

En la Figura 5-9 se ilustran los diagramas más comúnmente utilizados. Como puede observarse, la mayoría se usan tanto en el diseño estructurado como en el orientado a objetos.

Figura 5-9: Modelos para el Diseño Estructurado y el Diseño Orientado a Objetos



5-10: Elementos del diseño estructurado vs. elementos del diseño orientado a objetos

Los diagramas UML surgieron para el modelado orientado a objetos, sin embargo varios de éstos también se pueden utilizar para el modelado estructurado. En el capítulo de Modelado del software se estudiaron ya los diagramas de casos de uso, de clases y de secuencia.

La 5-10 ilustra los elementos del diseño estructurado y orientado a objetos en la construcción del software.

5.5 La descomposición (modularización) en el diseño

La descomposición según Braude (2003) consiste en desglosar un problema para que tenga las características de varios programas pequeños y sirve para afrontar la complejidad del problema. A la descomposición también se le llama *modularización.* En la programación estructurada los módulos son procedimientos y funciones que en conjunto proporcionan la funcionalidad del sistema, mientras que en la programación orientada a objetos los módulos son las clases que intervienen en la solución.

El principio del diseño modular consiste en descomponer el sistema en módulos. Cuando usamos el paradigma orientado a objetos, un módulo comúnmente consiste de un conjunto de clases e interfaces relacionadas. A nivel del diseño del sistema de software, se considera un módulo como un conjunto de clases e interfaces estrechamente relacionadas entre sí. A nivel del análisis, el módulo puede contener otros elementos tales como diagramas de casos de uso, etc.

5.5.1 La descomposición jerárquica

La descomposición jerárquica de un problema consiste en subdividir el problema vertical y horizontalmente. Como se ilustra en la Figura 5-11 con la subdivisión horizontal, la solución del problema se divide en varios sub-problemas, mientras que con la subdivisión vertical, se incrementan los detalles. En general, se van especificando los módulos necesarios para resolver cada sub-problema.

Durante la descomposición jerárquica se comienza por descomponer el problema inicial en cuatro o cinco módulos que cubren la totalidad de la solución (como se indica en la Figura 5-12) y posteriormente y en caso de ser necesario, éstos se descomponen para especificar más detalles. Este procedimiento se aplica una y otra vez hasta que se consigue resolver el problema por completo.

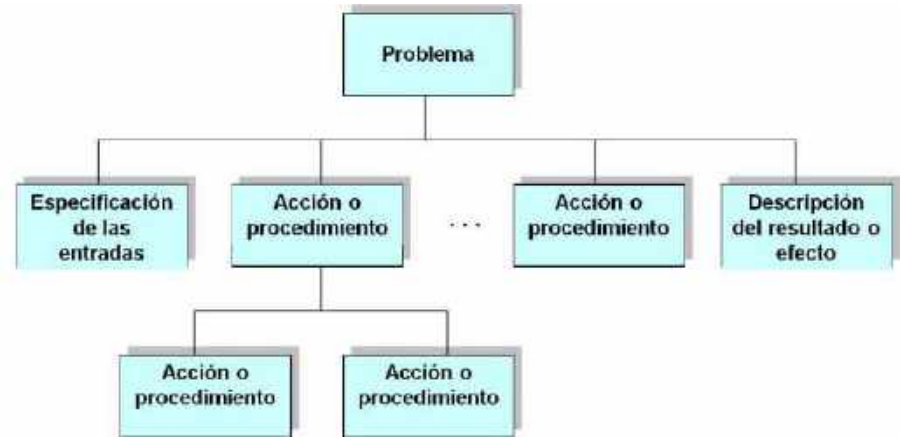


Figura 5-11: Descomposición jerárquica de un problema

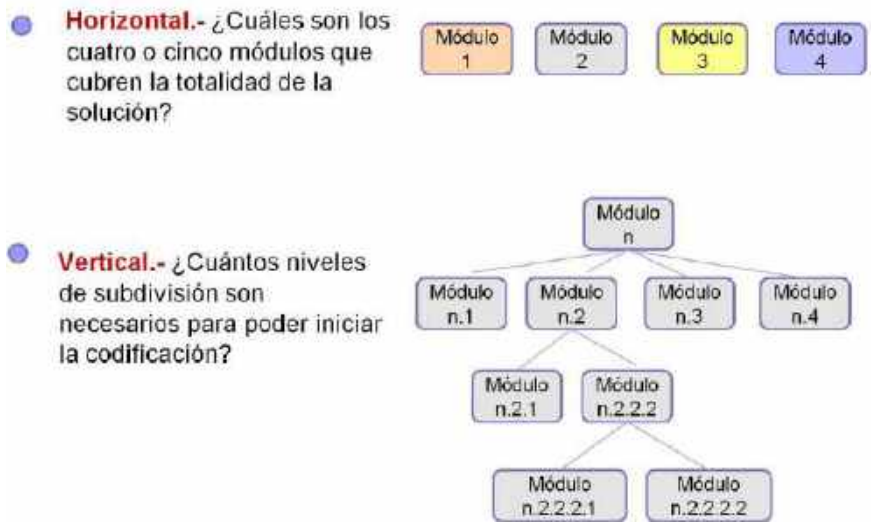


Figura 5-12: Descomposición jerárquica

La Figura 5-13 ilustra un ejemplo de descomposición jerárquica aplicado al diseño de un portal bancario.

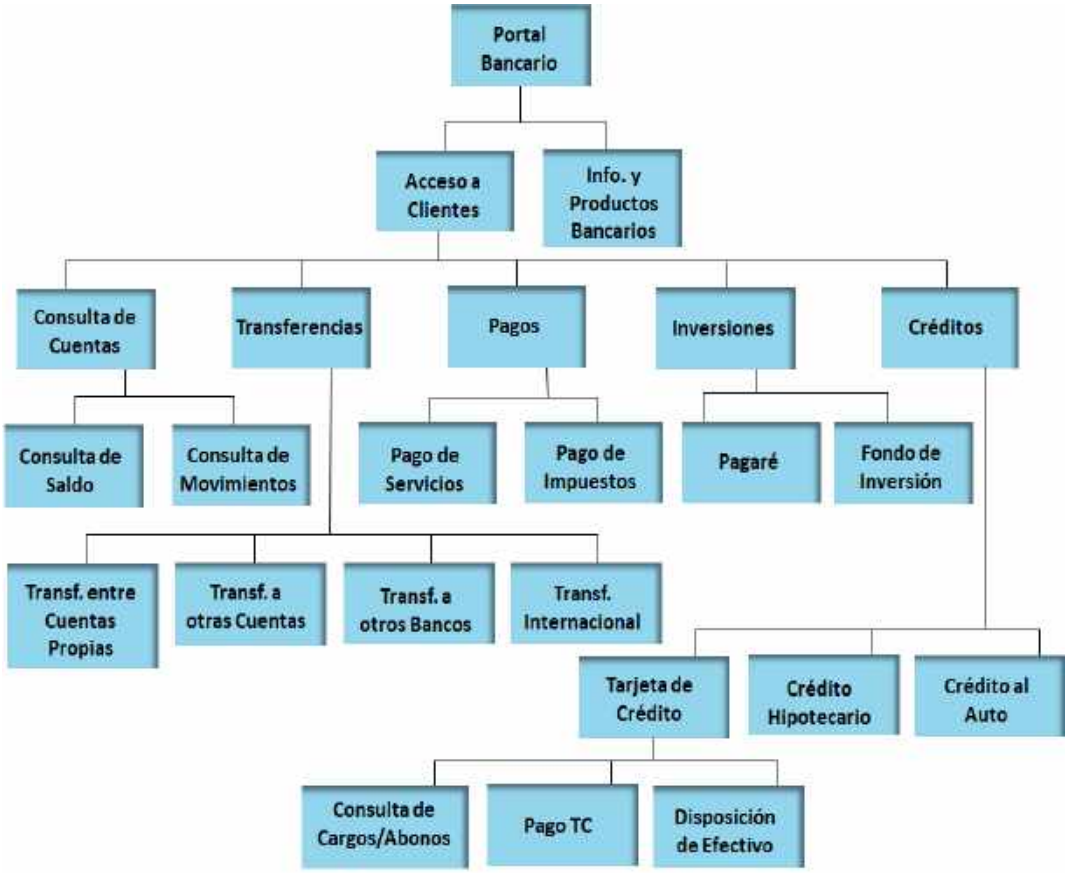


Figura 5-13: Ejemplo de descomposición jerárquica

5.5.2 Modularización efectiva

Hay dos aspectos a tomar en cuenta durante la modularización del software: la cohesión y el acoplamiento. La ***cohesión*** es el grado de relación que tienen entre sí las actividades que realiza un módulo. Es decir, el grado en el que un módulo se dedica a realizar un solo tipo de tareas. Y el ***acoplamiento*** es el grado en el que un módulo requiere comunicarse con otros módulos para realizar sus tareas. Como se ilustra en la Figura 5-14, para lograr una modularización efectiva se requiere que exista una *alta cohesión* en los módulos, es decir, cada módulo debe dedicarse, en la mayor medida posible, a realizar un mismo tipo de tareas. Además, se requiere que exista un *bajo acoplamiento*. Esto significa que los módulos deben hacer sus tareas de la forma más independiente que se pueda.

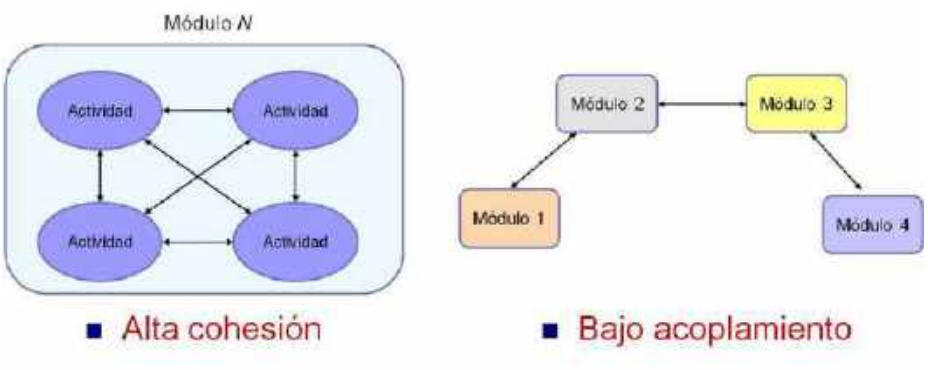


Figura 5-14: Modularización efectiva

Cuando se logra tener alta cohesión y bajo acoplamiento los cambios tienden a ser locales, es decir, los cambios se concentran en un módulo y los demás se ven poco afectados, lo que facilita el mantenimiento del sistema, como se observa en la Figura 5-15.

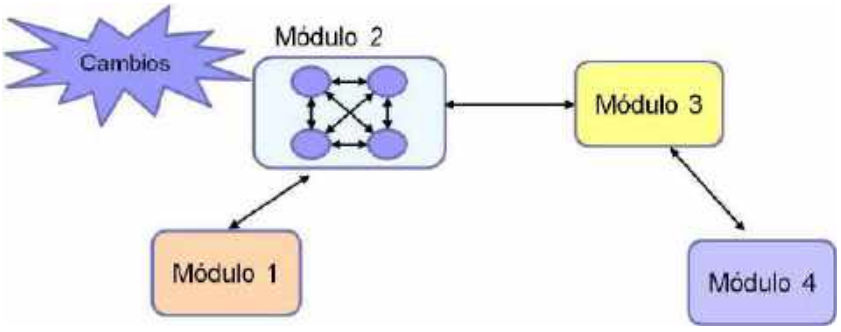


Figura 5-15: Mantenimiento cuando hay modularización efectiva

Lectura recomendada: "Heurística de diseño para una modularidad efectiva" [Pressman, 2010].

Entonces, existen dos criterios claves para valorar que tan apropiada es la propuesta de un diseño modular:

- Cohesión: se refiere al grado de relación entre los elementos (clases, interfaces, etc.) dentro de un mismo módulo.
- Acoplamiento (interdependencia): se refiere al grado de interrelación entre los diferentes módulos.

Un buen diseño modular implica una alta cohesión y un bajo acoplamiento.

La Figura 5-16 muestra un ejemplo ilustrativo de cohesión y acoplamiento entre clases y paquetes de clases, respectivamente, en un diseño orientado a objetos.

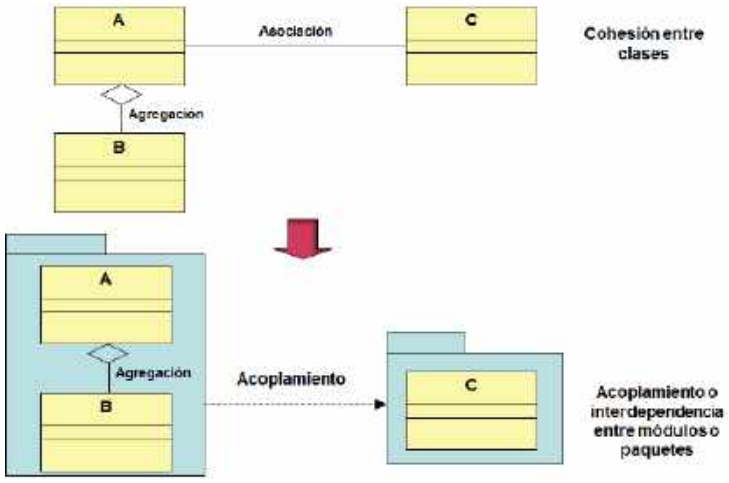


Figura 5-16: Ejemplo de cohesión y acoplamiento entre clases

5.6 Algunos términos usados durante el diseño

En esta sección definiremos dos términos que se usan comúnmente en el diseño, estos son: los *patrones de diseño* y los *artefactos de software*.

5.6.1 Los patrones de arquitectura y diseño

Un patrón de arquitectura o diseño es una solución que ha probado ser útil para resolver cierto tipo de problemas. Consiste en una combinación de componentes que resuelve un problema común de diseño. Los patrones de arquitectura o diseño se usan para aprovechar la experiencia de los diseñadores expertos y se logre un buen diseño más rápido que si se empieza desde cero. Un ***patrón de arquitectura o diseño*** es una solución a un problema en un contexto. El ***patrón es el resultado de la experiencia*** en un dominio específico. Un sistema bien estructurado utiliza patrones para aprovechar la experiencia de los expertos [Gamma et. al, 2003].

En ingeniería de software un patrón se refiere a la forma en la que un determinado problema de análisis, diseño, arquitectura, implementación, etc. fue solucionado, y cómo reutilizar la esencia de esta solución para resolver nuevos problemas. Un patrón encierra una experiencia, un conocimiento en la solución de problemas previos.

Un patrón:

- Maneja un problema de diseño (diseño arquitectónico o diseño de componentes) recurrente que aparece en situaciones de diseño específicas, presentando una solución a éste.
- Documenta experiencia de diseño existente que ha demostrado funcionar bien.
- Identifica y especifica abstracciones que están por encima del nivel de simples clases o instancias, o de componentes.

Además, un patrón proporciona un vocabulario y comprensión comunes para principios de diseño, es un medio para documentar arquitecturas de software, ayuda a construir software complejo a gran escala, y ayuda a manejar la complejidad del software.

En ingeniería de software un patrón es presentado como un esquema (*frame*) o descripción que consta de tres partes (ver Figura 5-17):

- **Contexto**: la situación donde surge el problema.
- **Problema**: descripción del problema que aparece repetidamente en el contexto dado.
- **Solución**: muestra una solución ya probada para el problema.

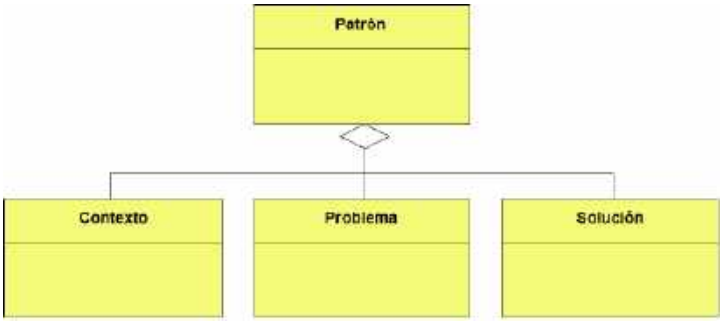


Figura 5-17: Aspectos de un patrón de diseño

Existe una gran variedad de patrones de arquitectura y diseño, los cuales se estudian en cursos más avanzados. En este curso explicaremos tres: Modelo-Vista-Controlador (patrón arquitectónico), fachada y *singleton* (patrones de diseño).

5.6.1.1 Modelo-Vista-Controlador (MVC)

Como se aprecia en la Figura 5-18, el patrón de diseño Modelo-Vista-Controlador (MVC) organiza la aplicación en tres partes independientes:

- ***La vista***.- Son los módulos involucrados en la interfaz con el usuario, por ejemplo, las páginas de internet que se despliegan en la computadora del usuario. La **vista** (***view***) es responsable de la interacción con el usuario. Presenta los datos en diferentes formas: puede haber más vistas en la medida en que sea necesario presentar los datos en formas diferentes. Recibe las entradas del usuario.

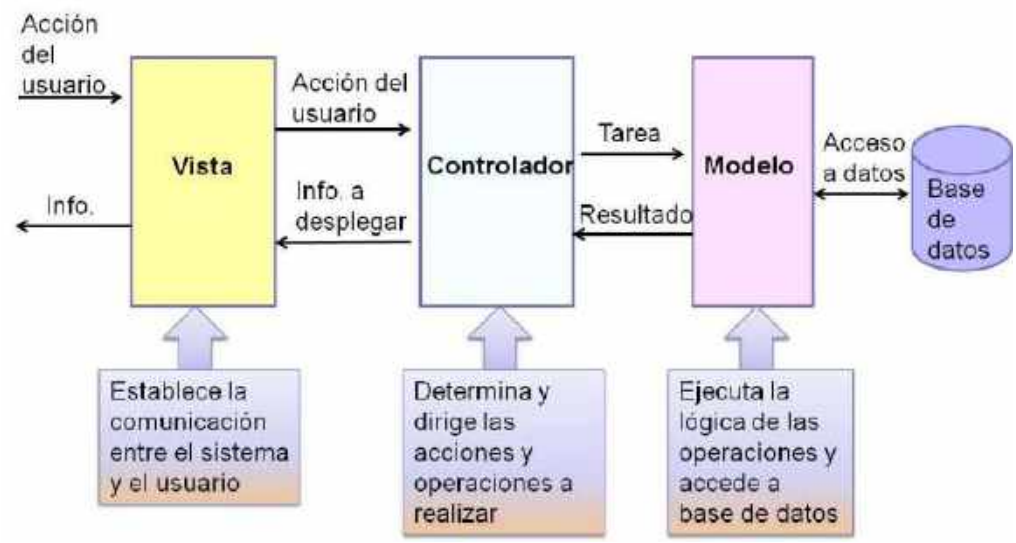


Figura 5-18: Modelo-Vista-Controlador (MVC)

- ***El controlador***.- Es el software que procesa las peticiones del usuario. Decide qué modulo tendrá el control para que ejecute la siguiente tarea. El **controlador** (***controller***) es sensible a las acciones del usuario, pudiendo recuperar los datos proporcionados por éste, trasladando los mismos en invocaciones de los métodos adecuados del **modelo** y seleccionando la **vista** apropiada (en base al estado y a la preferencia del usuario). En general, el **controlador** escucha los eventos de entrada que le llegan de la **vista**, y los relaciona con el **modelo**. Funge como coordinador entre la **vista** y el **modelo**.

- ***El modelo***.- Contiene el núcleo de la funcionalidad, es decir, ejecuta lo que se llama la *lógica del negocio*. Se le llama *lógica del negocio* a la forma en la que se procesa la información para generar los resultados esperados. El modelo se conecta a la base de datos para guardar y recuperar información.

El patrón MVC convierte la aplicación en un sistema modular, lo que facilita su desarrollo y mantenimiento. En la Figura 5-19 se ilustra la aplicación del patrón MVC al diseño arquitectónico del sitio web de un club deportivo virtual, cuya funcionalidad está dirigida al personal administrativo, entrenadores, socios y público en general.

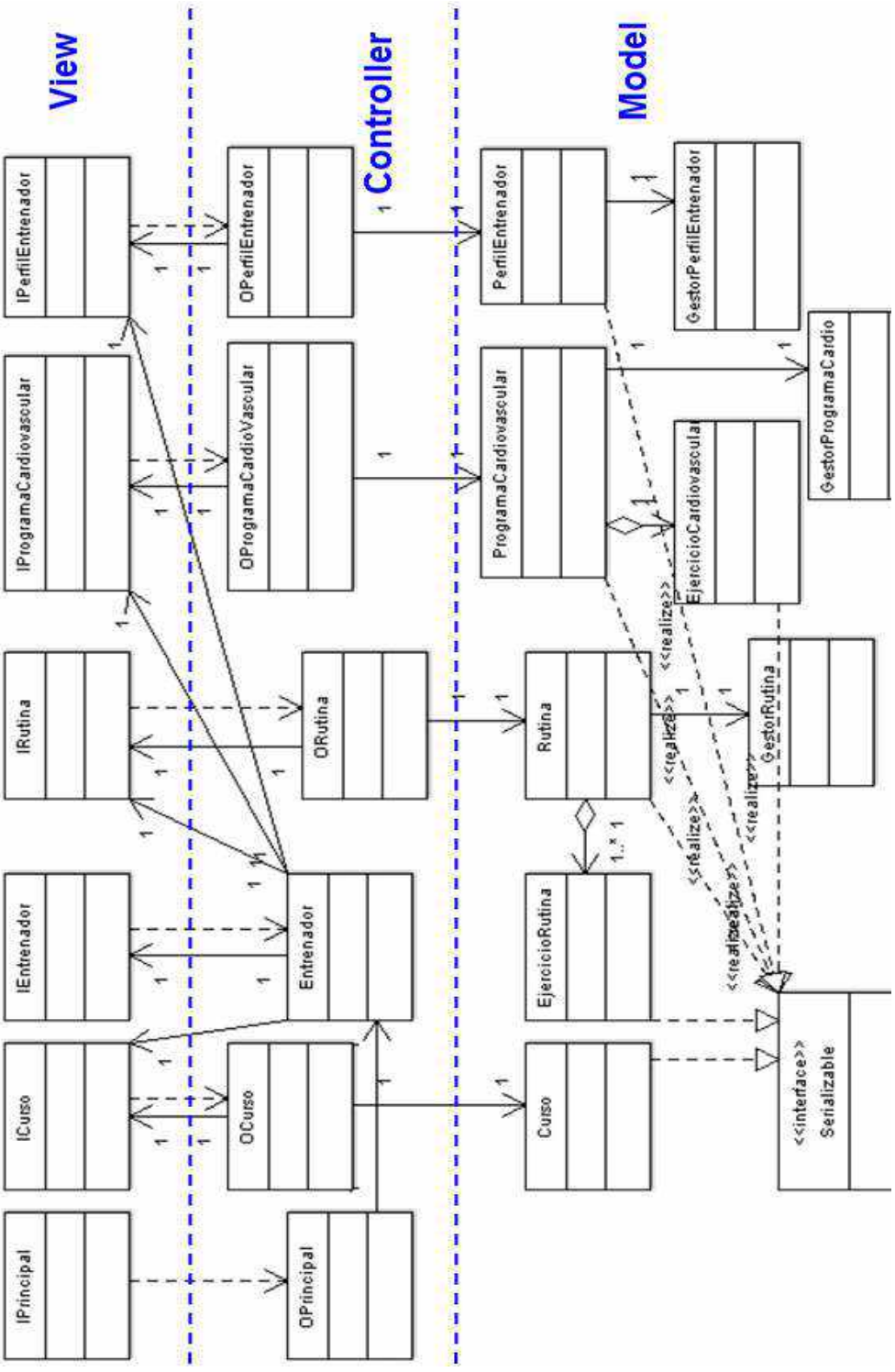


Figura 5-19: Ejemplo de MVC (diseño de un sitio web de un club deportivo)

5.6.1.2 *Singleton*

El patrón *Singleton* solo tiene sentido cuando se trabaja con el paradigma orientado a objetos. El *Singleton* restringe la creación de objetos. Así, cierta clase en particular solo se puede instanciar una sola vez con el objetivo de que el objeto creado concentre la información y los servicios que éste maneja (ver Figura 5-20). Con el singleton se evita el tener que coordinar la información en múltiples instancias de la clase lo cual puede ser muy complicado en algunos casos.

El patrón *Singleton* garantiza que una clase solo tenga una instancia, proporcionando un punto de acceso global a ésta. El patrón *Singleton* es comúnmente usado cuando: 1) Debe haber exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido, y 2) La única instancia puede ser extendida a través de la herencia, por lo que los clientes podrían utilizar la instancia extendida sin modificar su código. En el patrón *Singleton*, la propia clase es la responsable de su única instancia. La clase debe garantizar que no se pueda crear ninguna otra instancia. Una de las variantes más directas para utilizar el patrón *Singleton* en Java, es cuando declaramos la clase static, de forma tal que no sea posible instanciar objetos de la misma, fungiendo la propia clase como su única instancia.

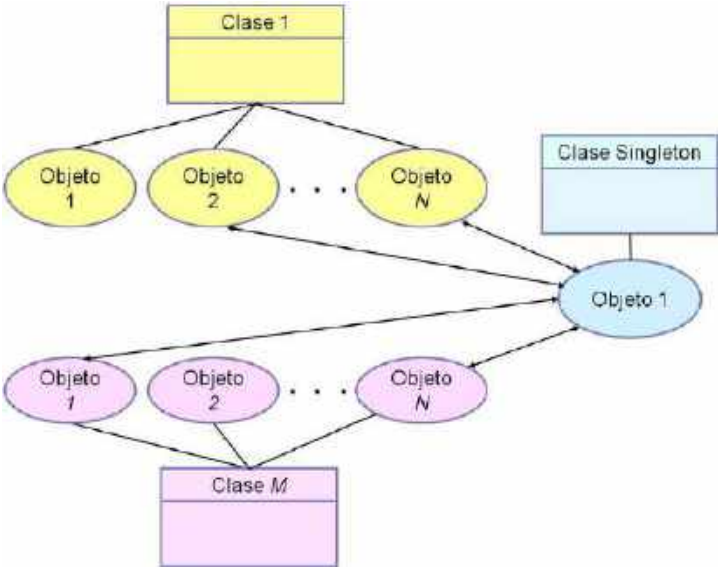


Figura 5-20: Singleton

La Figura 5-21 ilustra un ejemplo de aplicación del patrón *Singleton*, en el contexto de un portal bancario. Como se puede apreciar en este ejemplo, se desea que solo haya una *única instancia* de la clase cuenta (Singleton).

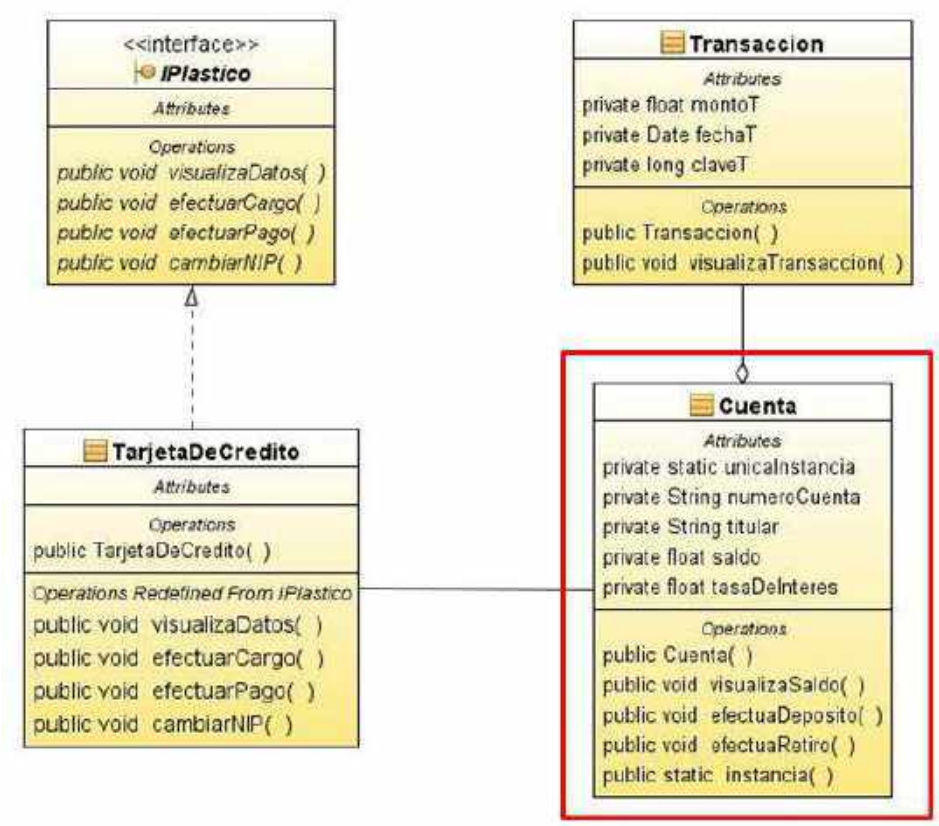


Figura 5-21: Ejemplo de un patrón singleton en un sistema bancario

5.6.1.3 Fachada

Con el patrón *Fachada* se simplifica la complejidad en el acceso a los servicios de un cierto grupo de módulos. El módulo *fachada* brinda una interfaz a este grupo de módulos que sea más simple y específica para la aplicación que se está creando, como se ilustra en la Figura 5-22.

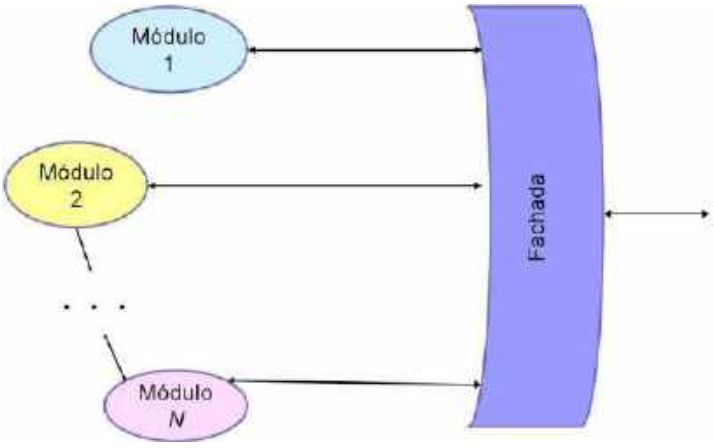


Figura 5-22: Fachada

El patrón Fachada (Gamma, E., 2003) proporciona una interfaz unificada para un conjunto de clases de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar. El patrón Fachada es comúnmente usado cuando: 1) Deseamos proporcionar una interfaz simple para un sistema complejo, 2) Exista muchas dependencias entre las clases clientes y las clases que implementan una abstracción, y 3) Necesitamos dividir en capas nuestros subsistemas, usando una fachada para definir un punto de entrada en cada nivel del subsistema.

Un ejemplo de aplicación de este patrón es la organización de una aplicación en subsistemas. Cada subsistema puede comunicarse con otros subsistemas a través de sus *fachadas.* Así, cada subsistema oculta su complejidad a los demás subsistemas a través de su fachada y se simplifica la comunicación entre todos los subsistemas de la aplicación.

La Figura 5-23 ofrece una representación genérica del uso del patrón Fachada.

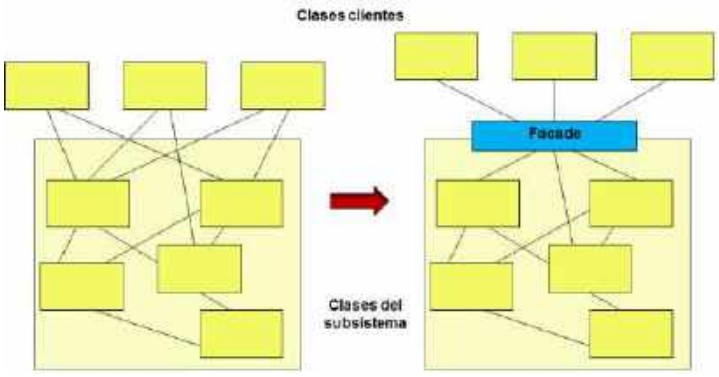


Figura 5-23: Representación genérica del uso del patrón Fachada

La Figura 5-24 ilustra una aplicación del patrón fachada para reducir la complejidad de las interacciones entre un grupo de clases de la Vista y un grupo de clases del Controlador, en un arquitectura MVC.

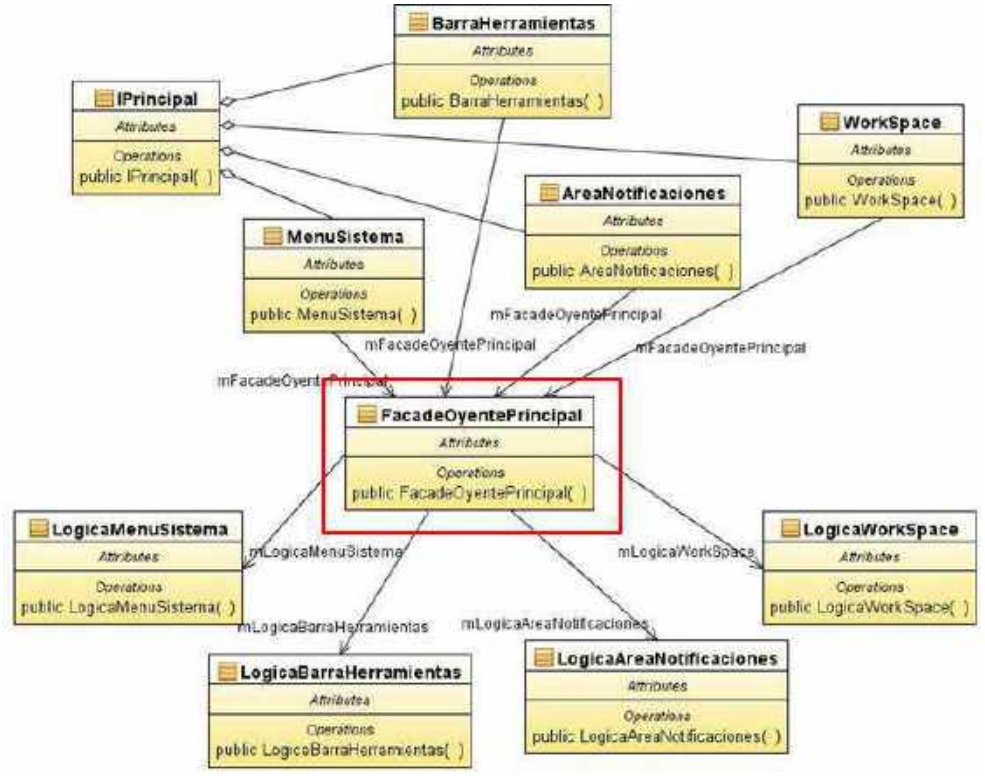


Figura 5-24: Fachada para reducción de complejidad de interacciones entre la vista y el controlador

5.6.2 Los artefactos de software

Un *artefacto* es un producto tangible resultante del proceso de desarrollo de software. Existen tres tipos de artefactos de software: un documento, un modelo y un elemento. En la Figura 5-25 se ilustran ejemplos de estos tres tipos de artefactos. El plan del proyecto, la especificación de requerimientos, el documento de diseño y el plan de pruebas son ejemplos de artefactos de software que consisten en un documento. Como ejemplos de artefactos de software que consisten en un modelo podemos mencionar: un diagrama entidad-relación de una base de datos, un caso de uso, un diagrama de clases, un diagrama de flujo, etc. Finalmente, como ejemplos de artefactos de software que son un elemento tenemos: una clase, un módulo compilado para hacer pruebas, el script de una base de datos, etc.

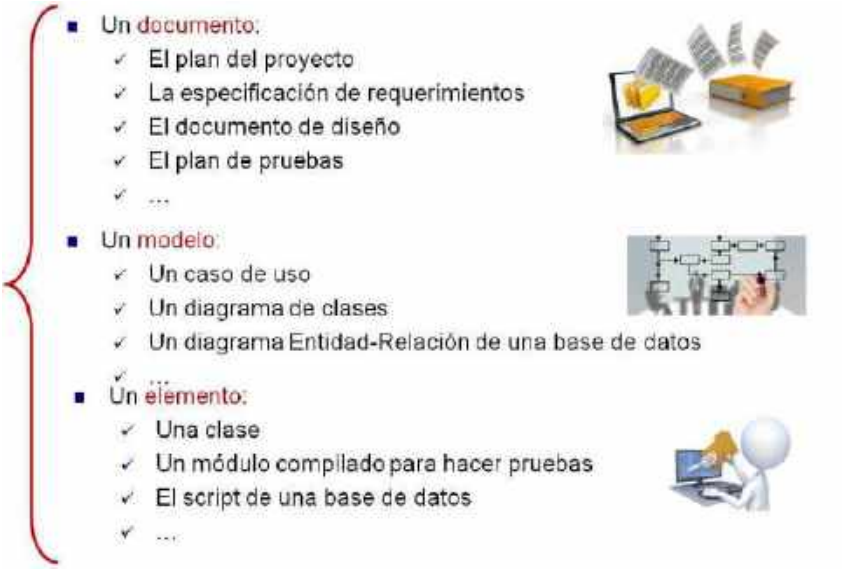


Figura 5-25: Artefactos de software

La Figura 5-26 ilustra un artefacto comúnmente utilizado en la etapa de especificación de requerimientos, uno de los diagramas de casos de usos del sitio web de un club deportivo virtual.

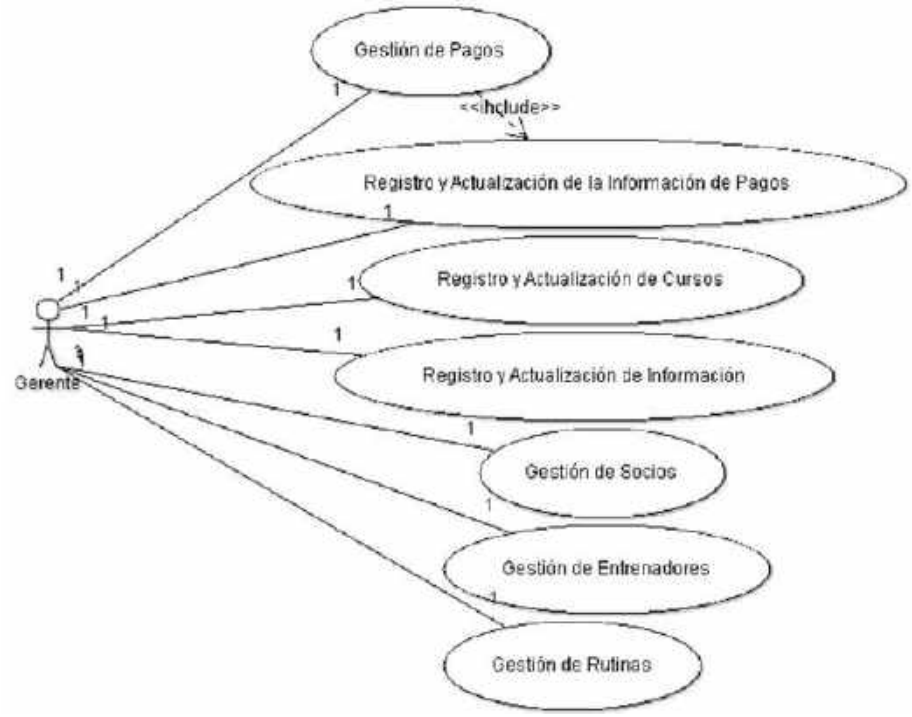


Figura 5-26: Ejemplo de artefacto (diagrama de casos de uso)

La Figura 5-27 ilustra el caso de un mal diseño de la interfaz con el usuario.

Figura 5-27: Fallas en el diseño [lapulgasnob.blogspot.com]

5.7 Resumen

El diseño es un proceso en el que se determina la estructura del sistema de software y de sus datos antes de iniciar su codificación. Además, proporciona en algunos casos la base de la lógica para codificar de tal manera que se cumpla con la especificación de requerimientos. Durante la etapa de diseño se produce el *Documento de Diseño* en el que se especifican: la arquitectura del sistema, el diseño global, el diseño de los componentes, el diseño de las interfaces de los componentes y el diseño de los datos.

Se considera que un diseño es bueno cuando es simple y comprensible, cuando es flexible, es decir que se puede adaptar a otras circunstancias agregando o modificando módulos y de preferencia que se pueda reutilizar, es decir, que se pueda utilizar en otros sistemas.

Tomando en cuenta el nivel de detalle, proceso de diseño se divide en dos fases: el *diseño de alto nivel*, también llamado diseño arquitectónico y el *diseño detallado***.** El diseño de alto nivel, define los módulos que componen al sistema y cómo éstos interactúan entre sí. Mientras que el diseño detallado define los datos del sistema sin llegar a los detalles de codificación. Los datos del sistema son los que se guardan en archivos o en bases de datos y los que se envían entre los módulos.

Teniendo en cuenta el enfoque, existe el *diseño estructurado* y el *diseño orientado a objetos.* El *diseño estructurado* toma en cuenta que el código consta de bloques lógicamente estructurados en forma de: instrucciones condicionales, ciclos y bloques lógicos. Mientras que el diseño orientado a objetos, supone que el código consta de objetos que interactúan entre sí.

A la descomposición o *modularización* consiste en desglosar un problema para que tenga las características de varios programas pequeños y sirve para afrontar la complejidad del problema.

5.8 Cuestionario

1.- Explica las características deseables de un sistema de software:

- ***Confiable***.-
- ***Robusto***.-
- ***Eficiente***.-

2.- Explica las actividades que se llevan a cabo durante:

- ***El diseño de alto nivel***.-
- ***El diseño detallado***.-

3.- ¿Cuál es la diferencia entre el *Diseño Estructurado* y el *Diseño Orientado a Objetos*?

4.- ¿Cuándo es útil el *Diseño Estructurado*?

5.- ¿Cuándo es útil el *Diseño Orientado a Objetos*?

6.- Explicar:

- ***Cohesión***.-
- ***Acoplamiento***.-

7.-¿Cuál es la ventaja de tener alta cohesión y bajo acoplamiento?

8.- ¿Qué es un patrón de diseño y cuáles son las ventajas de usarlo en el diseño?

9.- Menciona ejemplos de artefactos que sean: un documento, un modelo, un elemento.

6 La Codificación

6.1 Objetivos específicos del capítulo

- Comprender la utilidad de las reglas básicas de codificación y conocer ejemplos de éstas.
- Comprender la utilidad de la ejecución paso a paso de programas para la detección y depuración de defectos (*debugging*).
- Conocer qué es la refactorización.

6.2 La codificación

En ingeniería del software, la codificación se refiere a la traducción de la especificación del diseño a un lenguaje de programación. Como se ilustra en la Figura 6-1, la codificación traduce conceptos de diseño a programas fuente en un determinado lenguaje de programación (C++, Java, Python, etc.). La codificación es una fase del proceso de desarrollo de un sistema software, y en dependencia del ciclo de desarrollo seleccionado, la codificación podrá desarrollarse una vez que todos los componentes del sistema hayan sido diseñados, o en la medida que cada componente o grupo de componentes relacionados hayan sido diseñados.

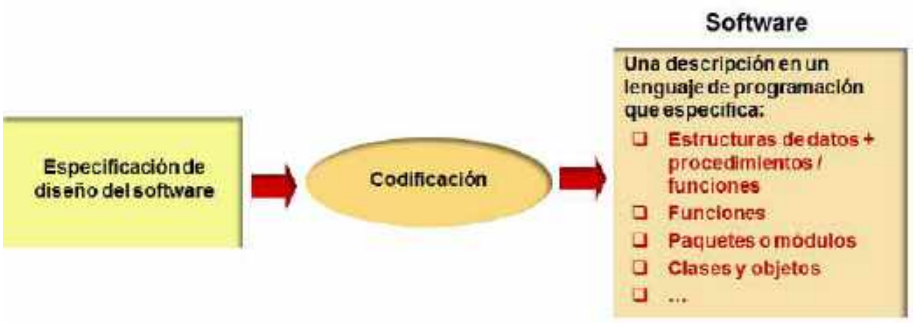


Figura 6-1: La codificación implementa el diseño en un lenguaje de programación

La codificación es la implementación del diseño del software en algún lenguaje de programación. Un diseño bien codificado debe tener ciertas características [Tsui et. al, 2014] proponen las siguientes metas de una buena codificación:

Código legible.- Otros programadores deben poder leer y entender el programa fácilmente.

Código trazable.- Un elemento o grupo de elementos de código deben corresponder a un elemento del diseño. Como el diseño debe corresponder a su vez con los requerimientos, entonces se debe poder trazar también desde un requerimiento hasta el código que lo implementa y viceversa.

Código correcto.- Cuando el código satisface algún requerimiento.

Código completo.- Se cumple con todos los requerimientos.

Buen desempeño (performance).- Un programa debe poder ejecutarse lo más rápido posible, siempre y cuando cumpla primero con todas las características anteriores.

Producir un código legible, que sea fácil de mantener es tan importante como que esté correcto y completo. En la gran mayoría de los casos la velocidad de ejecución es menos importante que la legibilidad del código.

6.2.1 El proceso de compilación

Una vez codificado, el programa debe pasar por el proceso de compilación en el que se verifica automáticamente que la sintaxis sea la correcta para luego producir el programa ejecutable. En la práctica se ha visto que es más sencillo arreglar los defectos de sintaxis poco a poco, que hacer todo el código de una sola vez y tener muchos defectos que corregir al final, pues hay defectos que generan más defectos y a veces no se sabe ni por dónde empezar (ver Figura 6-2 b).

Para lograr una compilación eficiente recomendamos compilar primero el "esqueleto del programa". En este primer paso se arreglan los problemas de configuración (librerías, rutas, etc.). Cuando el esqueleto está libre de defectos entonces agregar un pequeño bloque de código y compilar hasta que no haya defectos. Posteriormente agregar otro bloque y no pasar al siguiente hasta que la compilación sea exitosa. Continuar así hasta completar el programa, como se ilustra en la Figura 6-2 a).

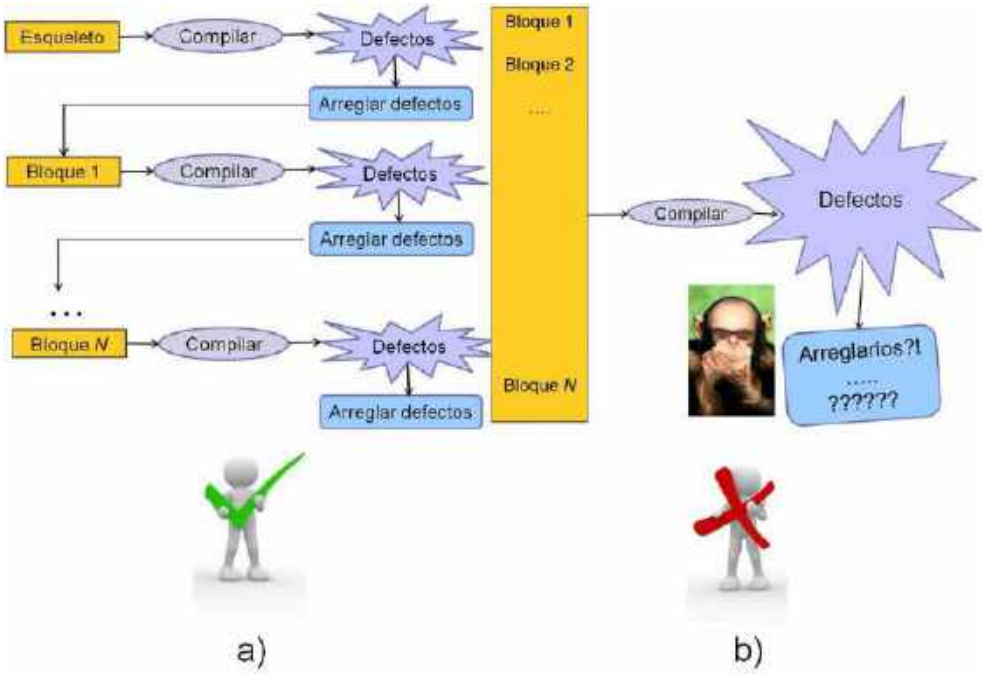


Figura 6-2: Buena y mala práctica de codificación-compilación

[JKUKE.COM]

El proceso de codificación se facilita si, además de usar el procedimiento anteriormente señalado, se respetan las reglas de codificación y se realiza una detección y depuración de defectos, lo que comúnmente se le conoce como "debuggear" (del inglés debugging). En las siguientes secciones estudiaremos las reglas de codificación y el proceso de *debugging*. Por otra parte, es importante hacer notar que un programa libre de errores de compilación ***no significa*** que *funcionará correctamente*.

Un programa debe estar hecho conforme a un diseño previo. Cuando el sistema es medianamente complejo y queremos codificar sin haber diseñado, lo más probable es que nos tardemos mucho más que el tiempo que se necesita para diseñar y codificar.

6.2.2 Reglas de Codificación

Las reglas de codificación definen un formato común para generar todos los programas con el objetivo de facilitar su lectura. Las reglas de codificación definen un *estilo de programar* que permita identificar fácilmente las estructuras en un programa. Por ejemplo: cuáles son las instrucciones que están contenidas en un ciclo u opción de una proposición condicional, las variables que se utilizan y su tipo, la explicación de lo que hace un bloque de código, etc.

Las reglas de codificación normalmente son definidas por una empresa en particular, pero también pueden hacerse por proyecto y para un lenguaje de programación específico.

6.2.3 Aspectos clave en las reglas de codificación

Los aspectos clave que las reglas de codificación deben definir son:

- Convenciones para nombres de identificadores (variables, constantes, clases, métodos y funciones)
- La declaración de identificadores (variables, constantes, clases, métodos y funciones)
- La indentación: Alineación de grupos de instrucciones en diferentes columnas según su uso
- Reglas para el tamaño y partido de líneas
- Reglas para comentarios explicativos
- Utilización de espacios en blanco

Además de estos aspectos generales hay otros aspectos que dependen de cada lenguaje en particular.

6.2.4 Ejemplo de reglas de codificación

A continuación, proporcionamos ejemplos de reglas de codificación para el lenguaje Java. Algunas de estas reglas son buenas prácticas que son válidas para cualquier lenguaje.

6.2.4.1 Convenciones para nombres de identificadores (variables, constantes, clases y métodos)

- El nombre debe ser *significativo*, es decir, debe describir el uso que se le da a una variable dentro del programa.
- Se debe evitar el uso de abreviaturas raras o letras solas, las letras solo se usan como contadores en un ciclo.
- Los nombres de clases e interfaces deben comenzar con mayúscula y los de objetos con minúscula.
- Usar la notación CamelCase, en la cual los nombres de variable que son compuestos deben llevar mayúscula al comienzo de cada palabra después de la primera, por ejemplo: `fechaInicio`, `claveMateria`, `ejemploDeVariable`, `promedioFinal`, `variableCamelCase`

6.2.4.2 Reglas para la declaración de identificadores (variables, constantes, clases y métodos)

- Cada variable debe declararse en una sola línea explicando su significado, por ejemplo:

```
int  level   // nivel de indentación
int  size    // tamaño de la tabla
```

- Las constantes deben ir con mayúsculas y alineadas. Por ejemplo

```
public static final int PRUEBAS        = 1;
public static final int INSPECCIONES   = 2;
public static final int REPORTES_ERROR = 3;
```

- Se debe evitar el uso de números directamente. Declarar una constante en lugar de usar números, por ejemplo:

```
perimetroCirculo = 3.1416 * radio       // Incorrecto

public static final double PI = 3.1416
perimetroCirculo = PI * radio           // Correcto
```

6.2.4.3 Reglas de indentación

- Los signos de igual (=) deben ir alineados cuando hay un grupo de asignaciones. Por ejemplo:

```
elControllerLogin   = new ControllerLogin( this, elView, elModel );
elControllerDoctos  = new ControllerDoctos( this, elView, elModel );
elControllerPruebas = new ControllerPruebas( this, elView, elModel );
```

- Alineación en el switch:
  - Cada `case` debe ir en un renglón aparte.
  - La instrucción `break` va en un renglón aparte.

- Las instrucciones deben ir todas alineadas en una misma columna y más indentadas que los `case`.

Ejemplo:

```
      switch( numInterfaz ) {
        case 1:
        case 2:
        case 4:  elViewLogin.presentarInterfaz( numInterfaz );
                 break;
        case 3:
        case 5:
        case 7:  elViewAdminProy.presentarInterfaz(
                    numInterfaz,
                    nombreProyActivo );
                 break;
      }
```

- Alineación en el `while` y `do-while`. Todo bloque de instrucciones debe ir más indentado que el bloque que lo contiene, ejemplos:

```
do {
    // bloque de instrucciones

}while( condicion )


while( condicion ) {
    // bloque de instrucciones

}
```

- Alineación en el `if` con un espacio antes de la llave:

```
if( condicion ) {
     // Bloque de instrucciones

}
else {

}
```

- Alineación en el for con espacios después de cada punto y coma:

```
for( int i=0; i<limiteSup; i++ ) {
     // Bloque de instrucciones

}
```

6.2.4.4 *Reglas para el tamaño y partido de líneas*

- El código no debe ocupar más de 80 columnas.
    - Cuando los parámetros de un método no caben en 80 columnas, se debe usar la notación alternativa con un parámetro en cada renglón. Por ejemplo:

```
public void guardarColores( ArrayList<colores> coloresList,
                            String producto ) {
```

```
        ...
}
```

- Cuando el acceso a campos o métodos de un objeto ocupe más de 80 columnas, usar la notación alternativa con cambio de línea en el punto el cual va alineado con el primer acceso en la línea anterior. Por ejemplo:

```
    elModel
    .guardarProyecto( integrantesList, nombre );
```

- Se debe evitar el partir expresiones matemáticas de manera confusa.

```
resultado = a * b / ( c + d - e )
            + 3 * f;                 // correcto

resultado = a * b / ( c + d
            - e ) + 3 * f;           // incorrecto
```

6.2.4.5 *Reglas para comentarios*

- Los comentarios explicativos o descriptivos deben ir al principio del bloque. Se deben escribir al mismo nivel de indentación que el del bloque que describen. Por ejemplo:

```
    /*
     *  Explicación para este bloque de instrucciones
     */
    for( int i=0; i <= n; i++ ) {
        . . .
```

- Los comentarios de una sola línea deben tener el siguiente formato:

```
/* Comentario en una sola línea */
```

- El formato de comentarios de fin de línea debe ser el siguiente:

```
   if( condicion ) { // explicar la condición
       instrucción;  // explicar la instrucción
```

6.2.4.6 *Reglas de utilización de espacios en blanco*

- En expresiones matemáticas y asignaciones, se debe dejar un espacio antes y después del signo igual y de los operadores. Por ejemplo:

```
      a = ( b + c - d ) / 2;   // Correcto

      a=(b+c-d)/2;             // Incorrecto
```

6.2.5 Otras reglas útiles

6.2.5.1 Anidamiento de instrucciones

- Se deben evitar los niveles de anidamiento en estructuras de control de más de tres niveles. Ejemplo

```
if( condicion1 )
   if( condicion2 )
       if( condicion3 )
           if( condicion4 )    // Incorrecto!

if( condicion1 )
   if( condicion2 )
       checarMasCondiciones( );  // Correcto!
```

6.2.5.2 Se deben checar primero las condiciones de excepción

Ejemplo de lo incorrecto:

```
public int metodoAlfa( ){
   if( condicion1 ) {
      if( condicion2 ) {
         /* acciones del método */
         . . .
         return 3;
      }
      else  return 2;
   }
   else return 1;
}
```

Ejemplo de lo correcto:

```
public int metodoAlfa( ){
   if( !condicion1 ) return 1;     // primer caso de excepción
   if( !condicion2 ) return 2;     // segundo caso de excepción
   /* pasó las pruebas de excepción */

   /* acciones del método */
   . . .
   return 3;
}
```

6.2.6 Prácticas y tips recomendables

Además de las reglas de codificación, hay prácticas recomendables que son independientes del lenguaje de programación. [Tsui et. al, 2014] recomiendan usar las librerías estándar en lugar de "reinventar la rueda". Las funciones de las librerías estándar están probadas y optimizadas, lo que hará que nuestro código sea rápido y tenga pocos defectos. También hay que probar cada función o método que codifiquemos antes de usarlo, así aseguramos que éste funciona correctamente. Además, hay que releer nuestro código y si es posible pedir a un colega que lo revise. La lectura de código es uno de los métodos más efectivos y simples para encontrar defectos en el software.

A continuación, enlistamos nuestras sugerencias.

- Reemplazar bloques de código similares por llamadas a una función común que tenga como parámetros variables que absorban las diferencias entre ellos.
- Usar paréntesis en expresiones para evitar ambigüedades.
- Dar a las variables de tipo booleano un nombre que indique condición, por ejemplo:

  ```
  esPrimo, estaIndefinido, huboError, seEncontro
  ```

- A los métodos de tipo void dar un nombre que sea un verbo, por ejemplo:

  ```
  calcularAlgo(), guardar(), imprimir(), capturar()
  ```

6.3 Ejecución del programa paso a paso para la detección y depuración de defectos en el código (debugging)

[5]Hacer *debugging* consiste en ejecutar un programa paso a paso con el objetivo de detectar y depurar los defectos del código. Detener la ejecución del programa en un punto deseado, al que se le llama ***breakpoint,*** permite conocer el contenido de las variables y estructuras de datos hasta este punto, lo que facilita en gran medida la detección de los defectos. Aunque *code debugging* se traduce al español como *depuración de código*, realmente no existe una palabra en español que signifique todo lo que implica la palabra *debug*. Es por eso que en español se acostumbra utilizar el término *debuggear* para indicar que se está haciendo el proceso de *debugging*. Hay defectos en el código que sería prácticamente imposible detectar si no existiera el proceso de debugging. [Tsui et. al, 2014] distinguen cuatro fases en el proceso de debugging.

1.- *Estabilización*.- En esta fase se encuentran las condiciones para reproducir el error. Es decir, cuáles son las entradas y estado del sistema que producen la falla que se quiere corregir.

2.- *Localización*.- Ésta puede llegar a ser la etapa más complicada. Consiste en localizar los defectos en el código que producen la falla. Es muy importante localizar correctamente la falla porque, si no, se corre el riesgo de hacer cambios que solo producirán más fallas.

3.- *Corrección*.- Consiste en cambiar el código en donde sea necesario para arreglar la falla. Cuando se entienden las causas de la falla, la corrección es relativamente sencilla. Sin embargo, puede ocurrir que aunque se detecta dónde se produce el error, no se sabe cuál es la causa. En este caso habrá que hacer una hipótesis, realizar los cambios que la hipótesis requiere y probarla o descartarla y comenzar nuevamente pero con una hipótesis distinta.

[5] http://intelmsl.com/wp-content/uploads/2012/04/balls.jpg

4.- *Verificación*.- Después de haber hecho las modificaciones que supuestamente arreglan la falla, será necesario comprobar que realmente se arregló para todos los casos (bajo condiciones diferentes) y además que no se hayan introducido nuevos defectos con estos cambios.

6.3.1 Ejemplo de debugging con NetBeans (Java)

Utilizaremos el debugger del IDE NetBeans y un programa en Java para ilustrar el proceso de *debugging*. El programa es muy sencillo, solo consiste en capturar un arreglo de enteros. La Figura 6-3 muestra un error al momento de ejecutar el programa.

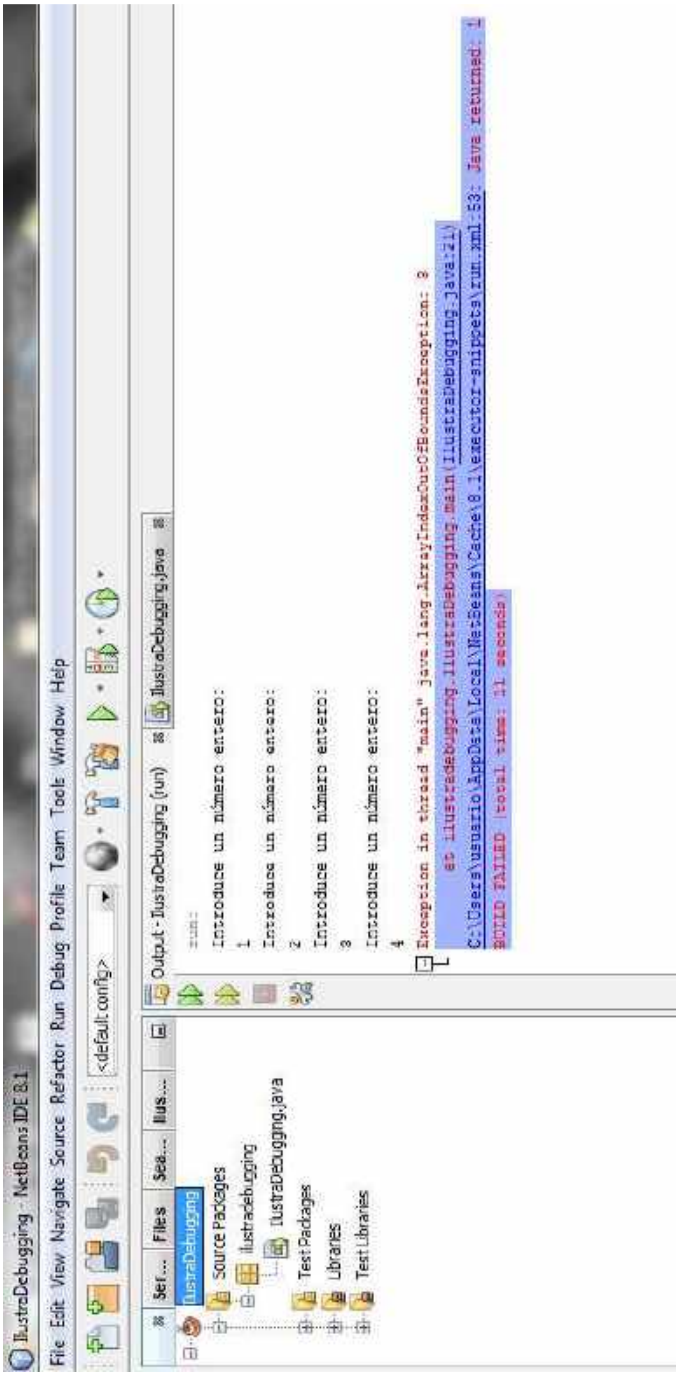


Figura 6-3: Error en la ejecución del programa ejemplo

El error que hemos introducido a propósito para ilustrar este ejemplo es muy sencillo. El mensaje de excepción que muestra la consola de ejecución proporciona información muy útil. Sin embargo, haremos un *debugging* para ilustrar la forma en la que se lleva a cabo la detección del error ejecutando el programa paso a paso. Lo primero que hay que hacer para *debuggear* es poner un ***breakpoint***, es decir, un punto en el que el programa haga un alto durante la ejecución. La manera de poner un breakpoint depende del IDE con el que se trabaje y por lo general es muy sencillo, en nuestro caso hemos hecho un clic del ratón en la línea donde queremos que el programa se detenga (ver Figura 6-4). Cuando hay un *breakpoint* en una línea de código, ésta queda resaltada con un color diferente (rosa o rojo en la mayoría de los IDE). Se pueden poner tantos *breakpoints* como se desee, pero es importante hacer notar que un *breakpoint* no se puede colocar en cualquier línea de código, ya que hay lugares en los que no está permitido. Por ejemplo, en el caso de Java, no está permitido poner *breakpoints* en las líneas de declaración de variables ni en las que solo contienen una llave.

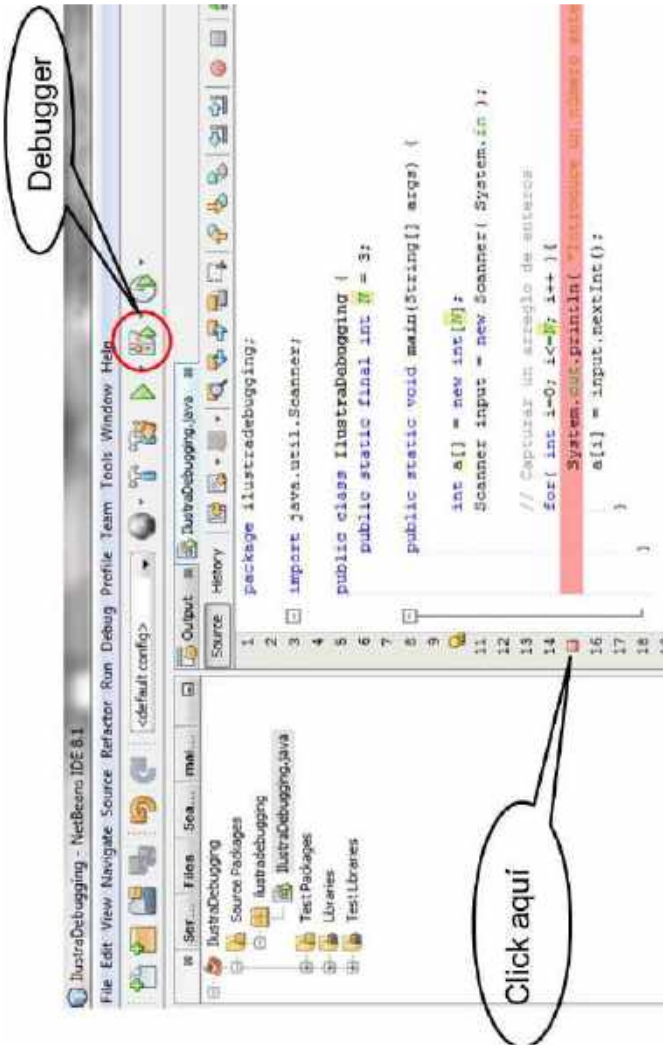


Figura 6-4: Breakpoint (lugar en donde el programa se detendrá)

Después de haber colocado un breakpoint en la o las líneas deseadas será necesario iniciar en modo *debugger*. En NetBeans, el botón del debugger está señalado con el círculo de la Figura 6-4. En la Figura 6-5 se observa que la ejecución del programa se detuvo en el breakpoint (renglón resaltado en rojo) y después se avanzó un paso (renglón resaltado en verde). Al seleccionar el botón con la flecha de avance (resaltada con un círculo en la figura) se puede ver que en el panel izquierdo aparecen las variables del *debugger*. Cuando se tiene un arreglo, como `a` en este caso, se puede ver el contenido de cada elemento dando un clic en el signo de "+" y cuando se tiene una variable sencilla, como `i`, se puede ver directamente su tipo y el valor que tiene, en la Figura 6-5 se aprecia que `i` es un entero y tiene valor "cero".

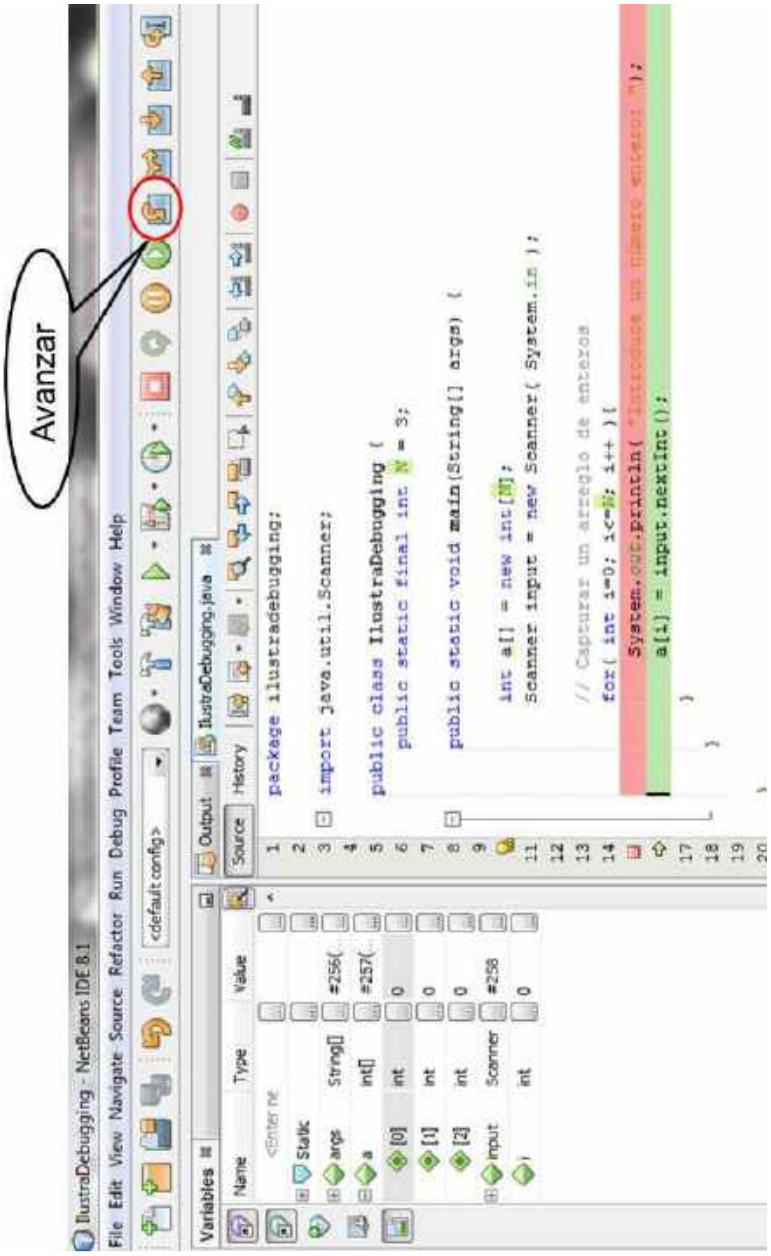


Figura 6-5: El programa se detuvo en el breakpoint

Cuando se está debuggeando un programa hay que alternar entre la pantalla de la consola de salida (output) y la del programa, ya que será necesario introducir los datos que el programa requiere para poder continuar. En la Figura 6-6 se observa que se introdujo el "1" en la consola de salida, y esto se ve reflejado en el primer elemento del arreglo `a`, además, el renglón resaltado en verde indica que ahora se comenzará con el siguiente ciclo del `for`.

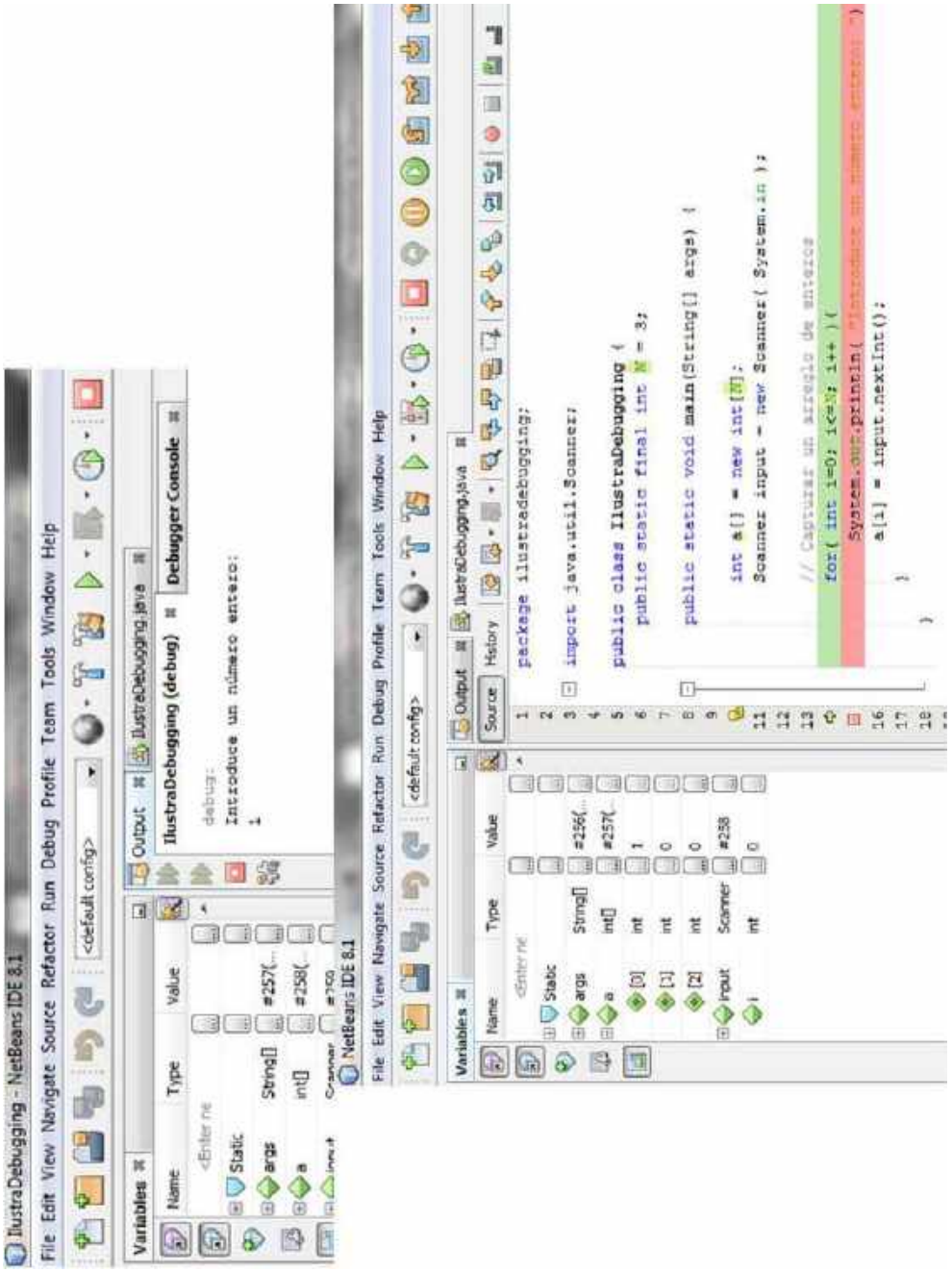


Figura 6-6: Datos después de ejecutar el primer ciclo del for

La Figura 6-7 muestra el estado del programa cuando ya se proporcionaron los 3 datos enteros. Como se puede observar, el arreglo `a` ya tiene sus datos completos, el contador `i` es 2, sin embargo, el programa está listo para continuar con un cuarto ciclo (reglón verde en la instrucción `for`). Esto generará una falla, ya que se estará accesando un cuarto lugar en el arreglo y éste no existe, pues se ha declarado el arreglo `a` de tamaño 3 (`N` = 3).

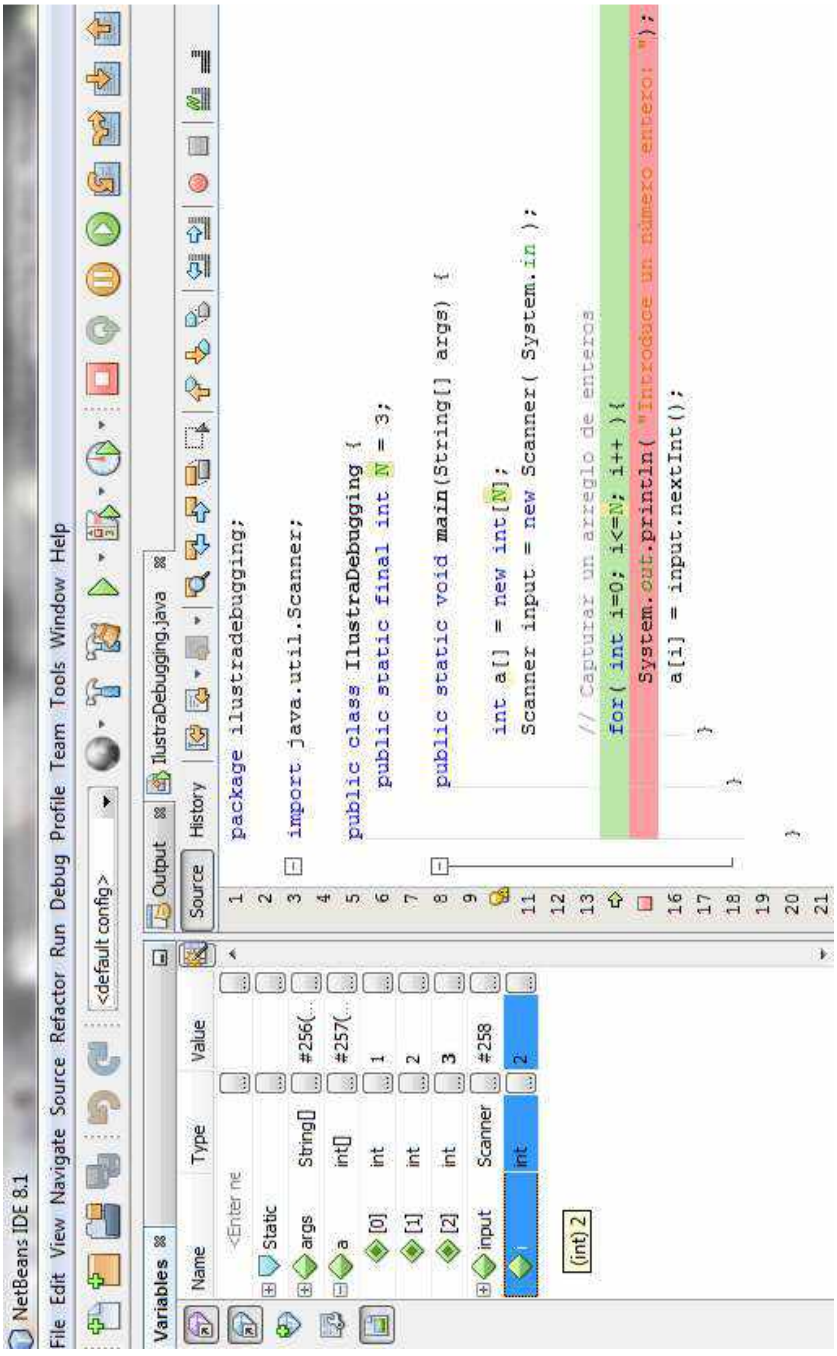


Figura 6-7: Estado del programa cuando el usuario ya proporcionó los 3 enteros

Verificamos que al ejecutar la cuarta iteración se presenta la falla. Si ponemos el cursor en la excepción e (ver Figura 6-8) se despliega la excepción que se lanzó.

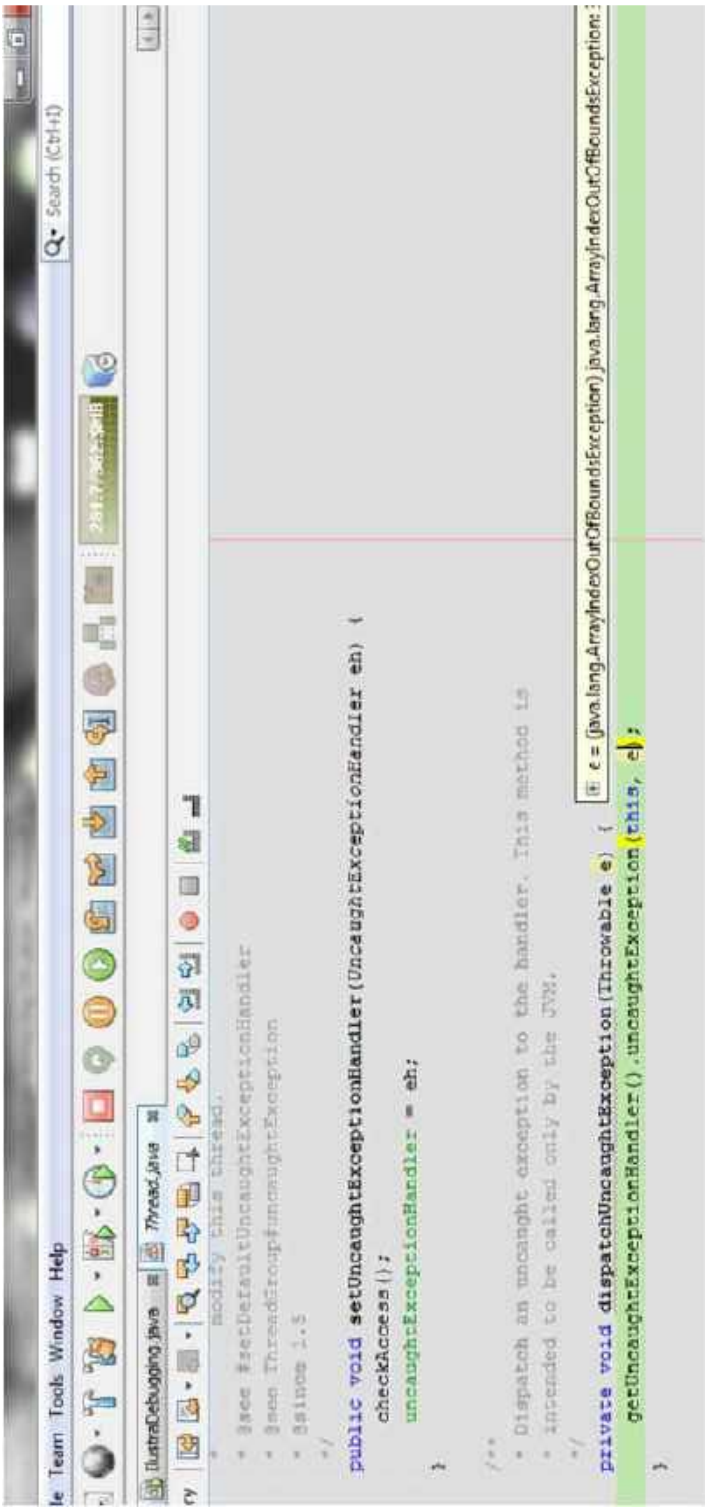


Figura 6-8: Se lanza una excepción

En este caso en particular `ArrayIndexOutOfBounds` indica que el índice del arreglo está fuera del límite. El defecto está en el ciclo `for` y para corregirlo es necesario cambiar:

```
for( int i=0; i<=N; i++ ){
```

por

```
for( int i=0; i< N; i++ ){
```

Si corremos nuevamente el programa podremos verificar que ya no se produce el error.

En el menú ***Debug*** que se encuentra en la barra de menús está la opción *New Watch…* la cual permite añadir cualquier otra variable del programa para poder visualizar su valor.

6.3.2 Diferentes formas de avanzar con el debugger

La Figura 6-9 ilustra cinco botones que hacen ejecutar el programa a diferentes ritmos durante una sesión de debugging. Estos botones también se pueden accesar con el menú ***Debug***. Con el primer botón de la izquierda solo se avanza al siguiente renglón. Con el segundo (de izquierda a derecha) el programa se ejecuta hasta el siguiente lugar en el que haya invocación a un método. Este botón es útil cuando en una misma expresión existen varias invocaciones a métodos. Mediante este botón se les puede trazar una por una. La tercera flecha sirve para llevar la ejecución paso a paso hacia dentro del método que se está invocando, una vez dentro del método se puede avanzar instrucción por instrucción (con el botón de la izquierda) y, en cualquier momento regresar al punto donde fue invocado usando el cuarto botón. El quinto botón (el de la derecha) se utiliza para ejecutar el programa hasta un punto deseado, en el cual situamos el cursor.

Figura 6-9: Botones del debugger

Si durante una sesión de debugging se quiere dejar correr el programa hasta el final, basta con seleccionar el botón de "run" (la flecha blanca dentro del círculo verde).

6.4 Refactorización de código

Es el proceso de modificar el código para que sea más claro y, por lo tanto, más fácil de mantener. Durante la refactorización no se corrigen defectos ni se agrega funcionalidad. Durante este proceso se hacen modificaciones y/o reestructuraciones en un programa sin alterar su comportamiento. *Refactorización* significa modificar el código de tal forma que la lógica de funcionamiento quede expresada lo más claramente posible, como *factorizar* en matemáticas (ver Figura 6-10).

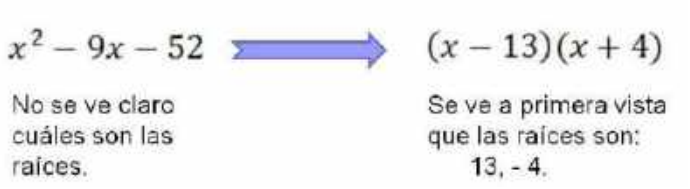


Figura 6-10: Refactorizar en matemáticas

A pesar de seguir las buenas prácticas y las reglas de codificación, es muy poco probable que se produzcan programas que no se puedan mejorar. Tomemos como ejemplo la elaboración de un documento de texto. Cuando el documento pasa por un proceso de revisión de estilo, se mejora su redacción y así las ideas que se quieren comunicar serán más claras. De igual manera, cuando un programa pasa por un proceso de revisión y modificación, al que llamamos *refactorización de código*, éste comunica mejor el proceso a los demás programadores.

Martin Fowler[Fowler, 1999] y Kent Beck[Beck, 1999] popularizaron el termino *code refactoring*, y es uno de los aspectos que se promueve en los métodos ágiles (estos métodos se estudian en el último capítulo de este libro). Como el estudio de las técnicas de refactorización es tema de otro curso, a continuación, solo mencionamos las principales acciones que se pueden llevar a cabo para mejorar el código durante la refactorización.

- Eliminar código muerto (el código que ya no se usa)
- Substituir código duplicado por el llamado a una función o método.
- Dividir una función/método excesivamente largo en métodos más pequeños dedicados a tareas sencillas.
- Substituir un algoritmo complicado por uno más sencillo y claro.

6.5 Resumen

En ingeniería del software, la codificación se refiere a la traducción de la especificación del diseño a un lenguaje de programación. La codificación traduce conceptos de diseño a programas fuente en un determinado lenguaje de programación.

Un buen código debe ser legible, trazable, correcto y completo. Además, debe elaborarse bajo *reglas de codificación* preestablecidas.

Hacer *debugging* consiste en ejecutar un programa paso a paso con el objetivo de detectar y depurar los defectos del código. Detener la ejecución del programa en un punto deseado, al que se le llama *breakpoint*, permite conocer el contenido de las variables y estructuras de datos hasta este punto, lo que facilita en gran medida la detección de los defectos.

6.6 Cuestionario

1.- ¿Cuáles son las características deseables de un buen código?

2.- Explicar qué significa que el código sea trazable.

3.- ¿Qué definen las reglas de codificación?

4.- Decir si es verdadero o falso

- ☐ Las reglas de codificación las puede hacer una empresa de software para que las sigan los desarrolladores que contrata.
- ☐ Las reglas de codificación se pueden hacer para que las sigan los que trabajan en un proyecto en particular.
- ☐ Las reglas de codificación se pueden hacer para un lenguaje en particular.

5.- Cambiar el siguiente código de tal forma que primero se chequen las condiciones de excepción.

```
public int checarPuntos( int puntos ){
   if( puntos > 0 ) {
      if( puntos > 100  ) {
          desplegarNivel( puntos );
          return 1;
      }
      else{
        System.out.println("Muy pocos puntos")
        return 0;
   }
   else{
     System.out.println("Cantidad inválida")
     return -1;
}
```

7.- ¿En qué consiste hacer *debugging*?

8.- ¿Qué es la refactorización?

7 Primer Caso de Estudio: Sistema I

7.1 Objetivos del capítulo

Objetivo general:

- Aplicar los conceptos estudiados en los capítulos anteriores, en el desarrollo de un sistema sencillo.

Objetivos específicos:

- Comprender, mediante un ejemplo práctico, la manera de redactar los requerimientos de un sistema de software.
- Conocer cómo se elaboran las pruebas de aceptación.
- Conocer, con un ejemplo, algunas de las herramientas que ayudan a elaborar un diseño.
- Comprender la conveniencia de diseñar antes de codificar.
- Codificar comenzando por el esqueleto y después añadiendo bloques.

7.2 Requerimientos del Sistema I

El ejemplo del desarrollo de un sistema muy sencillo al que llamaremos ***Sistema I***, permite visualizar la forma en la que se especifican los requerimientos, se elaboran las pruebas de aceptación, se elabora un diseño, y se codifica tomando como guía un diseño.

A continuación, presentamos la lista de requerimientos

R1.- El *Sistema I* no requiere interfaz gráfica (funciona en la consola).

R2.- Las operaciones que se pueden hacer con el *Sistema I* son las siguientes:

R2.1.- Ordenar tres enteros diferentes entre sí de menor a mayor.

R2.2.- Ordenar tres enteros de menor a mayor. Dos o los tres enteros pueden ser iguales.

Requerimientos de la interfaz de usuario.

R3**.-** Se presentará un menú con las siguientes opciones:

```
Selecciona la opción deseada para ordenar 3 números de
menor a mayor o salir:
      1.- Tres enteros diferentes entre sí
      2.- Tres enteros que pueden ser iguales
      3.- Salir
```

La Figura 7-1 muestra el diagrama de casos de uso del *Sistema I*.

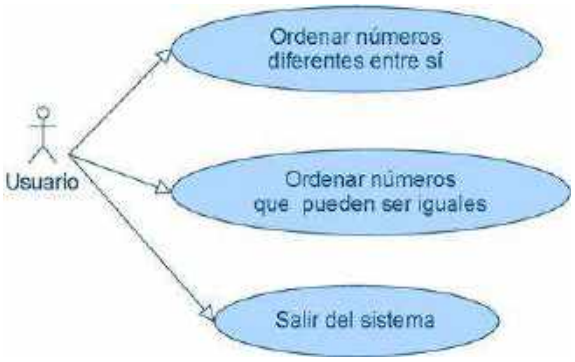


Figura 7-1: Diagrama de casos de uso del Sistema I

R4**.-** Cada vez que el sistema termina de ordenar los números proporcionados por el usuario, éstos se muestran en consola ordenados de menor a mayor y posteriormente se despliega de nuevo el menú de opciones.

R5**.-** El sistema terminará de ejecutarse cuando el usuario seleccione la opción "salir".

Requerimientos de los datos de entrada.

R6**.-** El usuario deberá proporcionar un entero correspondiente a una de las opciones del menú.

R7.- Si el usuario introduce un número diferente de 1, 2, o 3, se presentará nuevamente el menú, hasta que el usuario proporcione un número válido.

R8.- El sistema no debe permitir números reales como entrada.

R9.- Cuando se selecciona la opción 1, el sistema debe validar que los 3 números de entrada son diferentes entre sí para poder seguir con el ordenamiento.

Requerimientos de los datos de salida.

R10.- Sean *a, b* y *c* los datos proporcionados por el usuario. Una vez que se introducen 3 números válidos, éstos se ordenan y se despliegan en consola con el siguiente formato:

```
Los datos ordenados de menor a mayor son los siguientes: a, b, c
```

7.3 Pruebas de aceptación

Las pruebas de aceptación demuestran al cliente que el sistema sí funciona, en la mayoría de los casos, no son exhaustivas, es decir, no prueban todos los casos posibles.

A continuación, presentamos las pruebas de aceptación para las dos primeras opciones del *Sistema I*.

PA1.- *Pruebas de aceptación que demuestran que funciona la opción 1:*

PA1.1.- Proporcionar dos datos iguales: 3,3,5

Salida: El sistema debe pedir que los datos sean diferentes.

PAI.2.- Proporcionar dos datos iguales: 5,3,5

Salida: El sistema debe pedir que los datos sean diferentes.

PAI.3.- Proporcionar los 3 datos diferentes: 5,3,4

Salida: 3,4,5

PAI.4.- Proporcionar los 3 datos diferentes: 4,3,5

Salida: 3,4,5

PAI.5.- Proporcionar los 3 datos diferentes: 5,4,3

Salida: 3,4,5

PA2.- *Pruebas de aceptación que demuestran que funciona la opción 2:*

PA2.1.- II.1.- Proporcionar tres datos iguales: 5,5,5

Salida: 5, 5, 5

PA2.2.- Proporcionar dos datos iguales: 5,3,5

Salida: 3, 5, 5

PA2.3.- Proporcionar dos datos iguales: 5,5,3

Salida: 3, 5, 5

PA2.4.- Proporcionar los 3 datos diferentes: 5,3,4

Salida: 3,4,5

PA2.5.- Proporcionar los 3 datos diferentes: 4,3,5

Salida: 3,4,5

PA2.6.- Proporcionar los 3 datos diferentes: 5,4,3

Salida: 3,4,5

PA3.- *Prueba para verificar que no se permiten datos de entrada con punto decimal*

PA3.1.- En opción 1 o 2 proporcionar: 5, 7.3, 2

Salida: no debe permitir el 7.3

7.4 Diseño

Algunos podrían pensar que es más rápido saltarse el diseño y comenzar a codificar. Sin embargo, como ya se mencionó en el capítulo anterior, no diseñar trae complicaciones, pues es difícil que desde el código se tenga una visión global del funcionamiento del sistema. Si no diseñamos, lo más probable es que nos falte contemplar algunos casos especiales o condiciones de excepción. Además, será más complicado arreglar las fallas.

7.4.1 Diseño de la lógica del programa

Diseño de la opción 1

I.- *Los tres números de entrada deben ser diferentes entre sí*.- El diagrama de flujo de la Figura 7-2 muestra el funcionamiento de la lógica que permite validar que los tres datos de entrada sean diferentes entre sí.

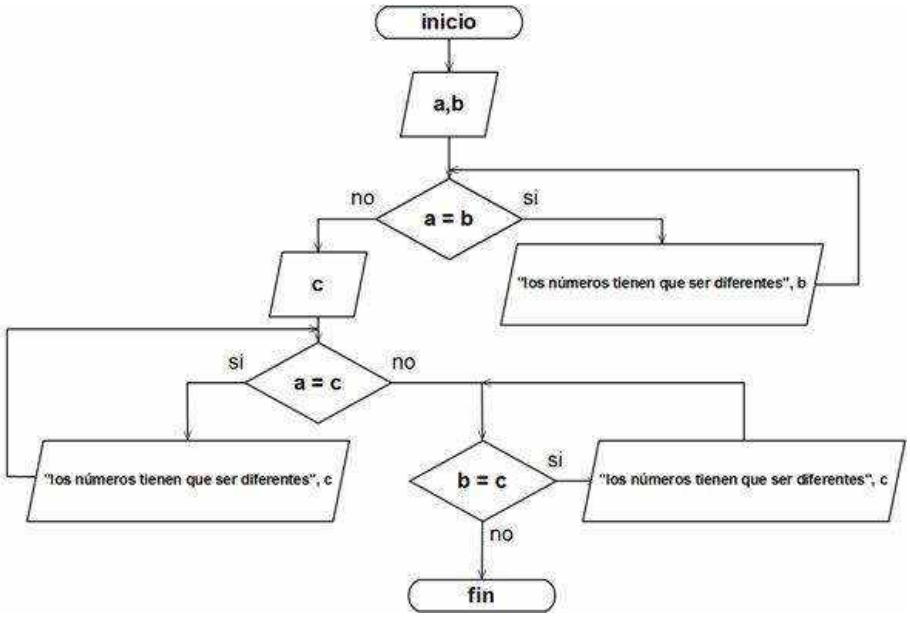


Figura 7-2: Diagrama de flujo para validar que los datos de entrada sean diferentes entre sí

II.- *Ordenamiento de menor a mayor de tres números diferentes entre sí*.- El diagrama de flujo de la Figura 7-3 muestra el funcionamiento de la lógica para ordenar los tres enteros de menor a mayor.

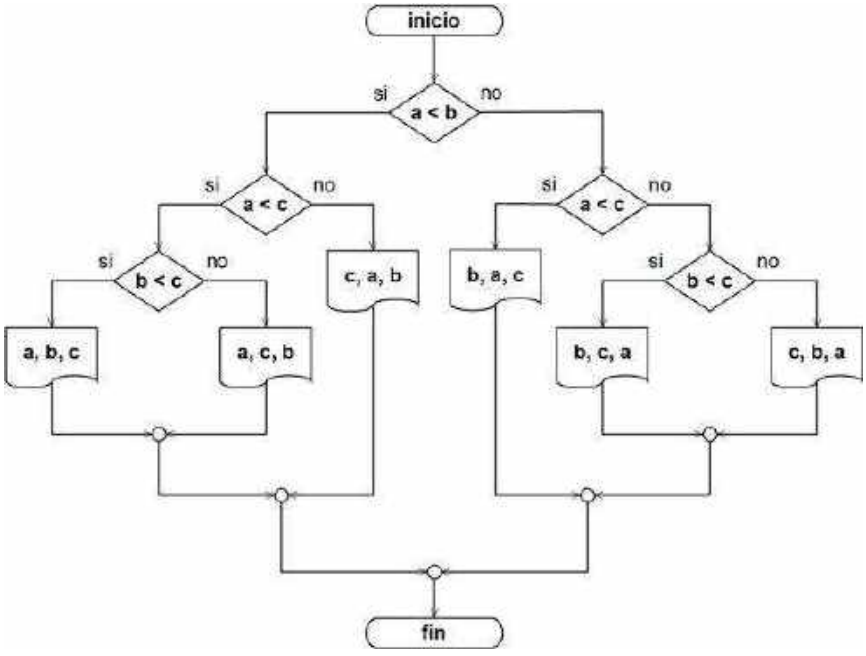


Figura 7-3: Diagrama de flujo para ordenar 3 enteros diferentes entre sí

III.- *Ordenamiento de menor a mayor de tres números que pueden ser iguales entre sí*.- El diagrama de flujo de la Figura 7-4 muestra el funcionamiento de la lógica para ordenar los tres enteros, que pueden ser iguales, de menor a mayor.

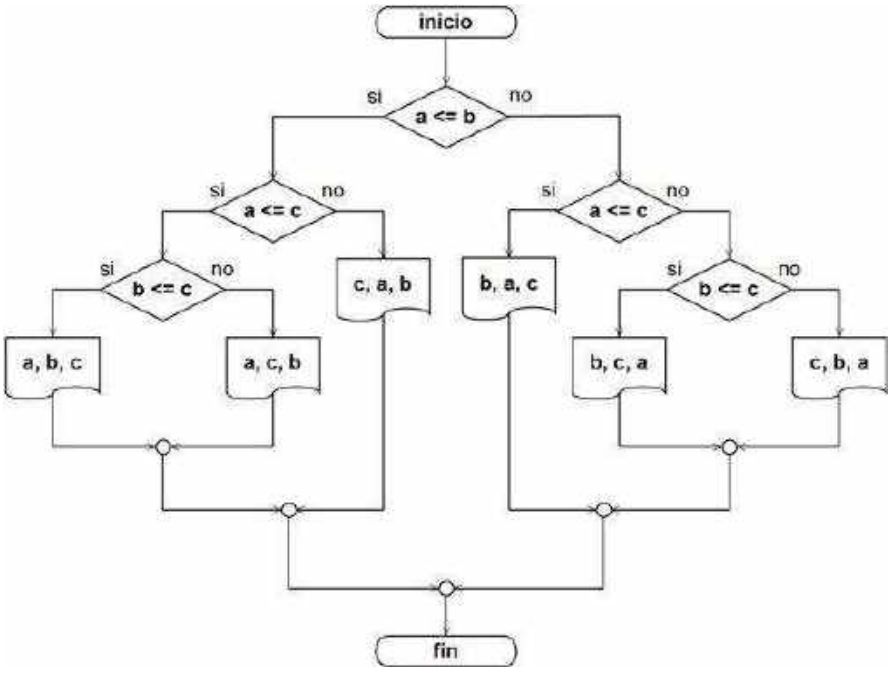


Figura 7-4: Diagrama de flujo para ordenar 3 enteros que pueden ser iguales entre sí

7.4.2 Diseño de las interfaces con el usuario

7.4.2.1 Formato de los datos de entrada

1.- El formato de los datos de entrada es el siguiente:

```
a = ?
b = ?
c = ?
```

2.- Para la opción 1 se despliega el mensaje:

```
Los números a, b y c deben ser enteros y diferentes entre sí
```

3.- Para la opción 2 se despliega el mensaje:

```
Los números a, b y c deben ser enteros
```

7.4.2.2 Formato de los datos de salida

Si los datos proporcionados son válidos, el sistema desplegará el resultado con el formato:

```
a, b, c.
```

Inmediatamente después se vuelve a mostrar el menú de inicio.

7.5 Codificación

7.5.1 Codificando el primer bloque

Codificamos el menú y los dos métodos vacíos (sin funcionalidad). El que ordena números diferentes: `ordenarNumsDif()` y el que también acepta números iguales: `ordenaNumsDif_o_Iguales()`. Como estos métodos aún no tienen código, será más fácil que el primer bloque no tenga errores de compilación. Además, se podrá probar el correcto funcionamiento del menú.

```
import java.util.Scanner;
public class SistemaI {
 /*
  *  Ordena de menor a mayor 3 números enteros
  */
  public static void main(String[] args) {

     int opcion;
     Scanner input = new Scanner( System.in );
     do{
      do{
         //  Menú
         System.out.println("Selecciona la opción deseada para ordenar 3"
                    + " números de menor a mayor o salir");
         System.out.println("1: Tres enteros diferentes entre sí");
         System.out.println("2: Tres enteros que pueden ser iguales");
         System.out.println("3: Salir");

         opcion = input.nextInt();  // lee la opción que elige el usuario

       }while ( ( opcion < 1 ) || ( opcion > 3 ));

       // El usuario elegió una opción válida
       switch( opcion ) {
             case 1:  ordenarNumsDif();
                         break;
             case 2:  ordenaNumsDif_o_Iguales();
                         break;
             case 3:  return;
         }
      }while( true );
    }

    private static void ordenarNumsDif(){

    }

    private static void ordenaNumsDif_o_Iguales() {

    }

  }
}
```

Falta codificar los métodos

La Figura 7-5 muestra que este primer bloque está libre de errores de compilación y funciona correctamente.

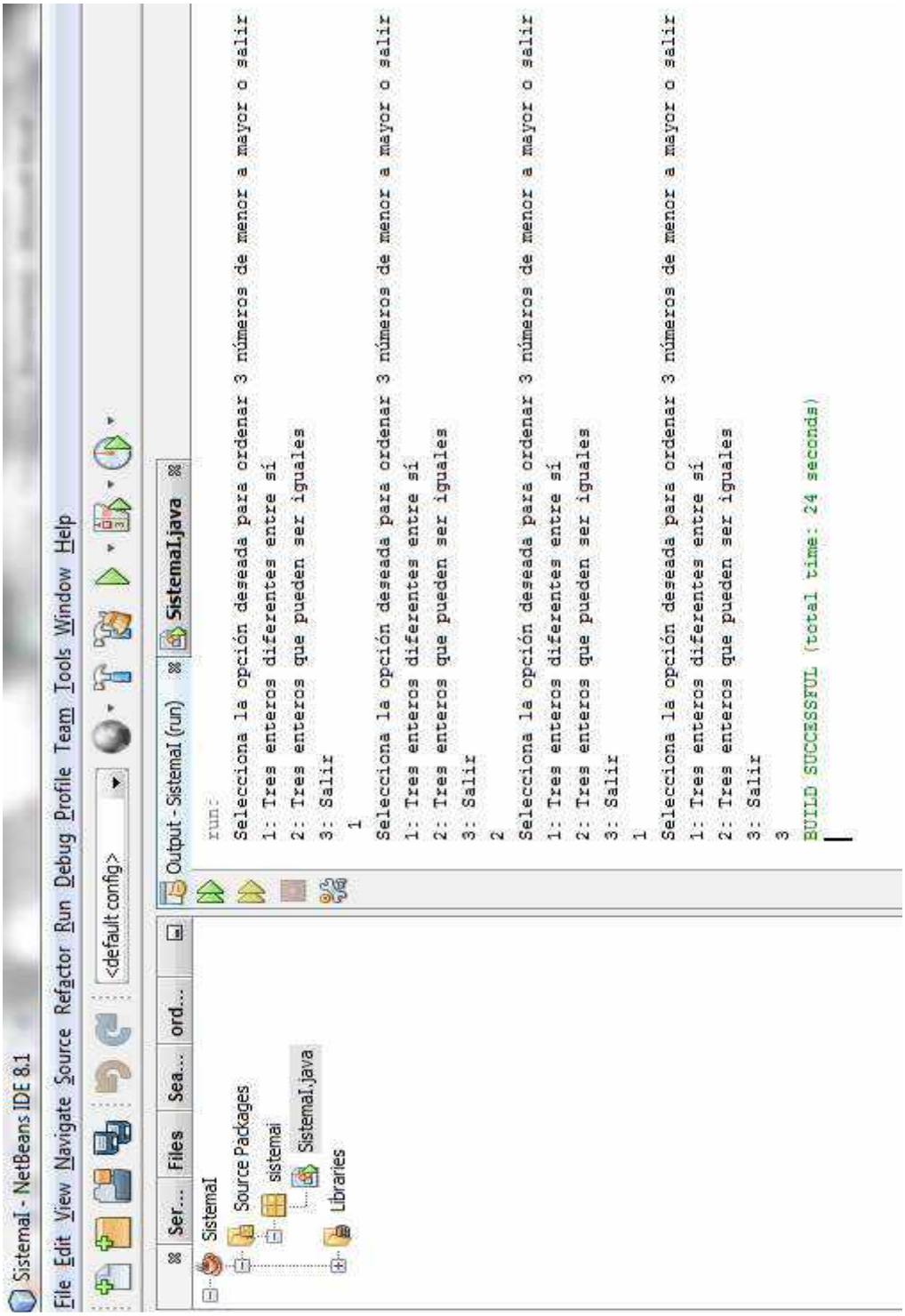


Figura 7-5: El primer bloque de código funciona correctamente

7.5.2 Codificando el segundo bloque

En el segundo bloque agregamos en el método: `ordenaNumsDif_o_Iguales()` la funcionalidad para capturar tres números enteros.

```
private static void ordenaNumsDif_o_Iguales() {
  int [ ] entradas;
  entradas = new int[3];
  Scanner input = new Scanner( System.in );

  // Pedir los 3 numeros enteros
  System.out.println( " Los números a, b y c deben ser enteros\n" +
                      " a = ?" );
  entradas[0] = input.nextInt();
  System.out.println( "  b = ? " );
  entradas[1] = input.nextInt();
  System.out.println( "  c = ? " );
  entradas[2] = input.nextInt();

  // Ordenar de menor a mayor
  . . .
}
```

Además en el método `ordenarNumsDif()` agregamos la funcionalidad para capturar tres números enteros que deben ser diferentes entre sí.

```
private static void ordenarNumsDif(){
  int [ ] entradas;
  entradas = new int[3];
  // Pedir entradas
  entradas = pideNumsDiferentes();
  // Ordenar de menor a mayor
  . . .

  return;
}
```

Para codificar `pideNumsDiferentes()` usamos como base el diagrama de flujo de la Figura 7-2.

```
private static int[] pideNumsDiferentes() {
  int a, b, c;
  int [ ] entradas;
  entradas = new int[3];
  Scanner input = new Scanner( System.in );
  System.out.println( " Los números a, b y c deben ser enteros "
                      + "diferentes entre sí\n" + "  a = ?" );
  a = input.nextInt();
  System.out.println( "  b = ? " );
  b = input.nextInt();

  if( a == b ){
      do{
         System.out.println( "Los números deben ser diferentes \n"
                             + " b = ? ");
         b = input.nextInt();
      }while ( a == b );
  }
  System.out.println( "  c = ? " );
```

```
  c = input.nextInt();
  if( a == c ){
      do{
         System.out.println( "Los números deben ser diferentes \n"
                             + " c = ? ");
         c = input.nextInt();
      }while ( a == c );
  }
  if( b == c ){
      do{
         System.out.println( "Los números deben ser diferentes \n"
                             + " c = ? ");
         c = input.nextInt();
      }while ( b == c );
  }
  entradas[0] = a;
  entradas[1] = b;
  entradas[2] = c;
  return entradas;
}
```

En la Figura 7-6 se muestra que el segundo bloque está libre de errores de compilación y que funciona correctamente.

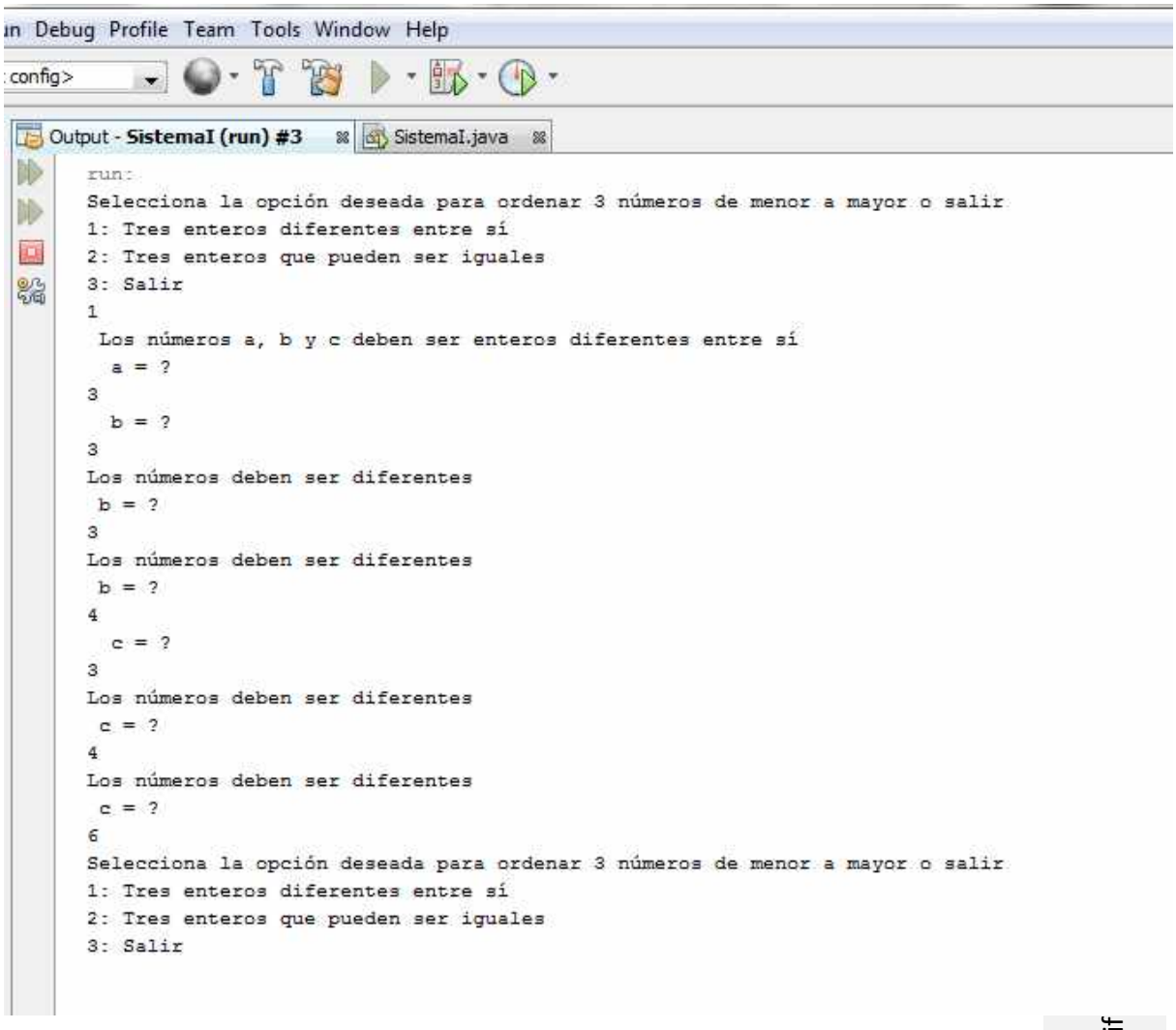


Figura 7-6: El segundo bloque de código funciona correctamente

7.5.3 Codificando el tercer bloque

Ahora agregaremos un tercer bloque en el cual el método `ordenarNumsDif()` contiene la funcionalidad para ordenar los tres números que son diferentes entre sí. Para codificar esta funcionalidad nos basamos en el diagrama de la Figura 7-3.

```
private static void ordenarNumsDif(){
  int a, b, c;
  int [ ] entradas;
  entradas = new int[3];
  //Pedir entradas
  entradas = pideNumsDiferentes();
  a = entradas[0];
  b = entradas[1];
  c = entradas[2];

  // Ordenar de menor a mayor
  if( a < b ){
    if( a < c ){
```

```
      if( b < c )
          System.out.println( a + ", " + b + ", " + c );
      else
          System.out.println( a + ", " + c + ", " + b );
    }else
      System.out.println( c + ", " + a + ", " + b );
  }else{
    if( a < c )
        System.out.println( b + ", " + a + ", " + c );
    else{
      if( b < c )
          System.out.println( b + ", " + c + ", " + a );
      else
          System.out.println( c + ", " + b + ", " + a );
    }
  }
  return;
}
```

En la Figura 7-7 se muestra que el tercer bloque está libre de errores de compilación y que funciona correctamente.

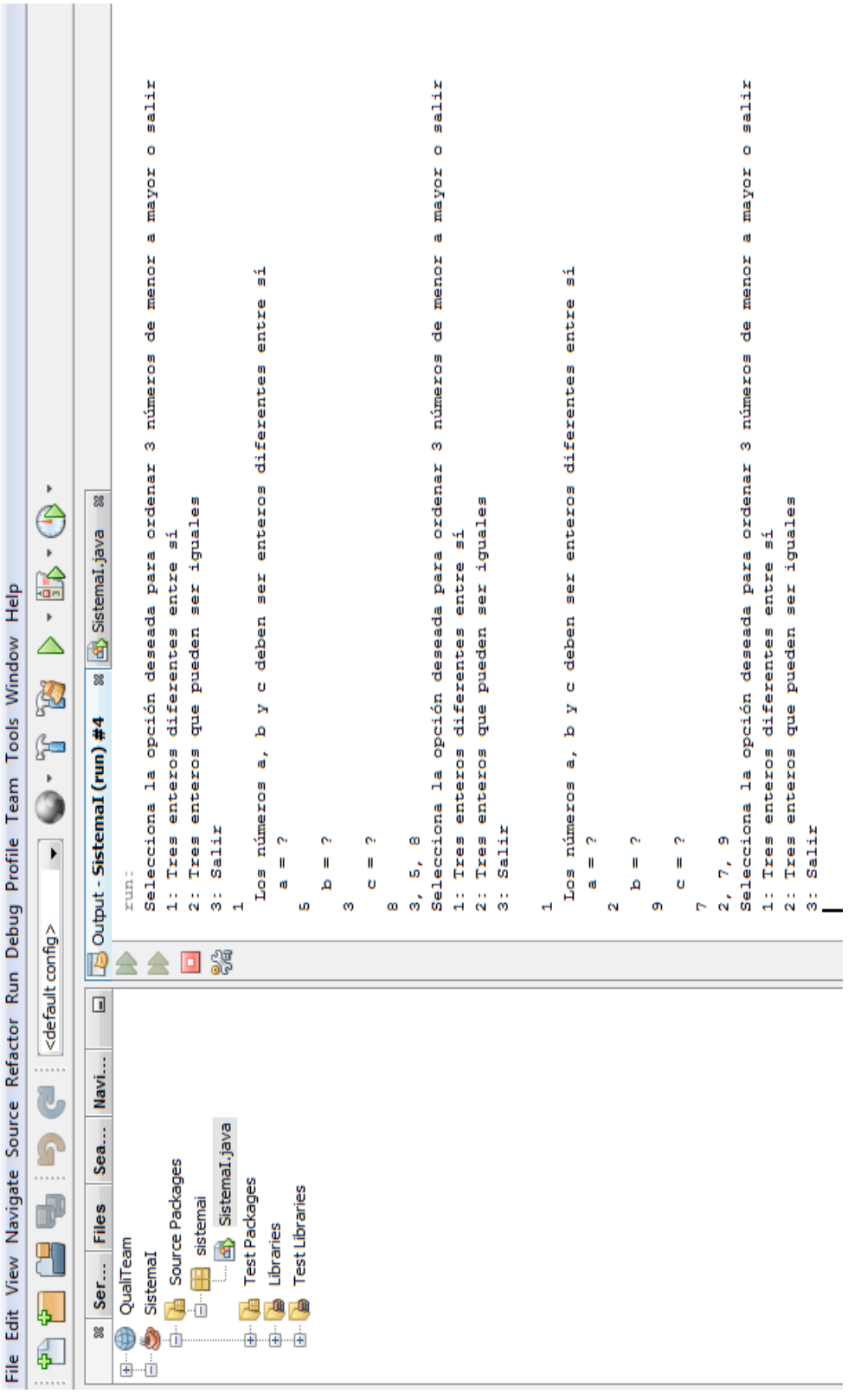


Figura 7-7: El tercer bloque funciona correctamente

7.5.4 Codificando el cuarto bloque

Finalmente para el cuarto bloque agregamos en `ordenaNumsDif_o_iguales()` la funcionalidad para ordenar los tres números que pueden ser iguales o diferentes entre sí. Actividad: codificar esta funcionalidad en base al diagrama de la Figura 7-4.

```
private static void ordenaNumsDif_o_Iguales() {
   int a, b, c;
   Scanner input = new Scanner( System.in );

   // Pedir los 3 números enteros
   System.out.println( " Los números a, b y c deben ser enteros\n" +
                       " a = ?" );
   a = input.nextInt();
   System.out.println( "  b = ? " );
   b = input.nextInt();
   System.out.println( "  c = ? " );
   c = input.nextInt();

      // Ordenar de menor a mayor
      . . .

}
```

Usar el diagrama de flujo de la fig. 7-4 para codificar

En la Figura 7-8 se muestra que el cuarto bloque está libre de errores de compilación y que funciona correctamente.

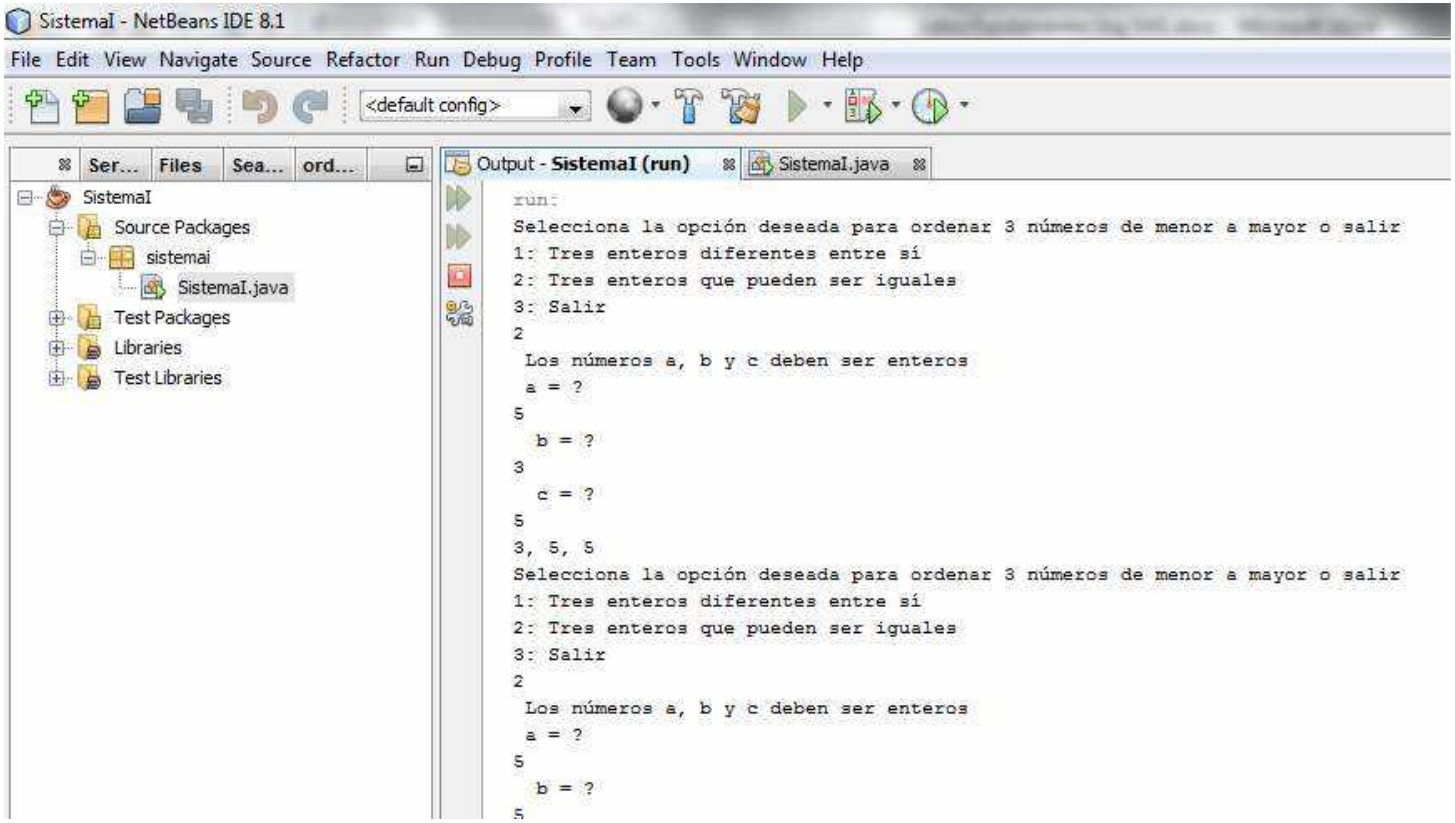


Figura 7-8: El cuarto bloque funciona correctamente

8 Calidad, Pruebas, Gestión de la Configuración y Mantenimiento del software

8.1 Objetivos específicos del capítulo

- Comprender el concepto de calidad en el software y su importancia.
- Conocer las principales actividades de un sistema de control de calidad del software.
- Comprender el significado de los principales términos que se manejan en el ámbito de calidad del software.
- Conocer en qué consiste el proceso de pruebas.
- Conocer en qué consiste la gestión de la configuración.
- Conocer cuáles son las actividades fundamentales que se llevan a cabo durante el mantenimiento.

8.2 Calidad en el software

8.2.1 Introducción a calidad del software

La definición de calidad es subjetiva y depende del enfoque que se le dé. Por ejemplo, cuando nos enfocamos solamente en el producto, se dice que un software es de calidad si tiene buenas características, independientemente de lo que un cliente o usuario pida. Sin embargo, esta definición no toma en cuenta el grado de satisfacción del cliente, quien finalmente pagará por el producto. Bob Mirmant, fundador y director de la empresa Eliant Inc. dice que "la calidad es solo una percepción, y solo los clientes pueden definirla". Si tomamos en cuenta al producto y al cliente podemos decir que un sistema de software es de calidad cuando cumple con los requerimientos especificados y sirve para el propósito para el que fue creado.

No obstante, a la subjetividad presente en el concepto que encierra la calidad del software podemos establecer que desarrollar “software de calidad” significa que:

- El sistema software desarrollado satisface plenamente los requerimientos funcionales especificados, así como las expectativas y necesidades del cliente.
- Tanto el diseño como la implementación corresponden de manera precisa a los requerimientos funcionales especificados.
- Los requerimientos no funcionales especificados – tales como, portabilidad, seguridad, eficiencia, reusabilidad y extensibilidad, etc. – han sido considerados durante el desarrollo del sistema software.
- El sistema software desarrollado es correcto, eficiente, flexible, confiable y seguro.
- La reusabilidad y extensibilidad del sistema software desarrollado han sido garantizadas.
- La calidad ha sido asegurada, es decir, ha sido medida y controlada, a través de cada una de las fases de desarrollo del sistema software: análisis de requerimientos, diseño arquitectónico, diseño de componentes, implementación, etc.

Entre los principales atributos de calidad del software, se encuentran los siguientes:

- Fiabilidad
- Flexibilidad
- Seguridad
- Robustez
- Adaptabilidad
- Portabilidad
- Modularidad
- Usabilidad
- Reutilización
- Protección
- Eficacia

Producir software de calidad es importante porque las fallas que se producen en él pueden tener consecuencias graves, como provocar grandes pérdidas económicas, o incluso provocar la muerte de alguna o muchas personas. Los sistemas computarizados en los bancos y en las casas de bolsa son ejemplos en donde una falla de software puede generar grandes pérdidas de dinero. La tolerancia a fallas depende del sistema que se esté construyendo. En los hospitales, en transporte aéreo, en sistemas de seguridad, en sistemas automatizados que manejan herramienta o material peligroso existe software que, en caso de fallar, puede provocar tragedias que pueden costar muchas vidas humanas. En ese tipo de sistemas, es muy importante ofrecer una garantía de que el software no tendrá fallas o por lo menos que, si se presentan, éstas no serán graves.

Además de lo anterior, minimizar la cantidad de defectos en el software creado es importante porque se reducen los costos de su producción. En la práctica se ha observado que un 50% del costo de corrección de defectos encontrados en la fase de pruebas se deriva de errores humanos cometidos en fases de desarrollo mucho más tempranas. Si los defectos se detectaran en fases tempranas, no sería necesario repetir una gran cantidad de pruebas ni hacer correcciones en todas las fases subsecuentes. Esto indica claramente que vale mucho la pena hacer un esfuerzo por corregir lo antes posible los defectos introducidos por los errores cometidos. Para lograr producir sistemas de software que tengan la menor cantidad posible de defectos, las empresas establecen un sistema de ***gestión de calidad.***

La ***gestión de calidad*** es el conjunto de actividades destinadas a la detección y corrección temprana de defectos en el sistema de software que se está produciendo.

Como se establece en [Sommerville, 2007] y como se ilustra en la Figura 8-1, la gestión de la calidad del software abarca tres actividades fundamentales:

- Garantía de la calidad.

- Planificación de la calidad.
- Control de la calidad.

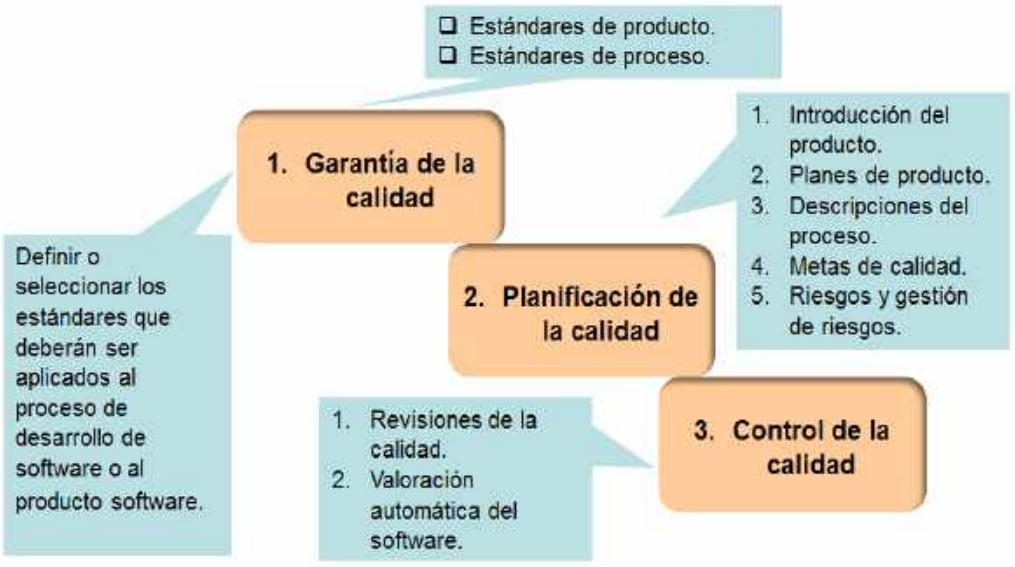


Figura 8-1: Gestión de la calidad del software

Cada empresa establece su sistema de control de calidad de acuerdo con sus necesidades y características. En todo sistema de control de calidad se hace una evaluación para verificar la existencia y calidad de los productos en cada etapa del desarrollo del sistema. A continuación, presentamos una lista con los artefactos que comúnmente se evalúan en cada una de las fases del desarrollo de un proyecto:

- Durante el *Inicio del proyecto*:
  - El plan del proyecto
  - El plan de gestión de configuraciones
  - El plan de aseguramiento de la calidad

- Durante el *Análisis y Especificación de Requerimientos*:
  - Especificación de Requerimientos
  - Descripción de las pruebas de aceptación

- Durante el *diseño*:
  - Documento de diseño (de alto nivel y detallado)
  - El plan de pruebas
  - Manual del operador
  - Manual del usuario

- Durante la codificación:
  - Código fuente
- Durante las *pruebas unitarias* (de módulo):
  - Procedimientos de las pruebas unitarias
  - Resultados de las pruebas unitarias
- Durante las *pruebas del sistema*:
  - Procedimientos de las pruebas de integración
  - Resultados de las pruebas de integración
  - Reportes de defectos
  - Trazado de los reportes de defectos al código corregido
- Durante la *gestión de la configuración*:
  - Solicitud de cambio
  - Implantación del cambio
  - Reporte de revisión del cambio
  - Reporte de distribución del cambio

En la sección 8.4 se estudia la gestión de la configuración.

Las principales actividades que se llevan a cabo para el control de calidad son:

- ***El proceso de revisión entre colegas***.- Consiste en someter a revisión un artefacto (documento de especificación, de diseño, de pruebas, código, etc.) para que otros colegas retroalimenten al autor con sus observaciones. El autor modifica su artefacto hasta llegar a un consenso con los demás colegas.
- ***Las inspecciones del software***.- Es una técnica formal de revisión de los artefactos del sistema (documentos y código) en la que se revisa la consistencia entre éstos y se verifica que se cumpla con las *buenas prácticas*. Estas buenas prácticas están previamente definidas y documentadas.
- ***Las pruebas del software***.- Consisten en la ejecución del sistema haciendo uso de la funcionalidad para la que fue diseñado, con objeto de verificar que la funcionalidad se ejecuta correctamente.

8.2.2 Diferencia entre Error, Defecto y Falla

En el ámbito de la calidad del software las palabras *error*, *defecto* y *falla* tienen un significado específico:

- **Error**.- Es una acción incorrecta de un ser humano que produce un defecto en un artefacto.

- **Defecto**.- Es un elemento de un artefacto que es inconsistente con sus requerimientos y que eventualmente producirá fallas.

- **Falla**.- Es una discrepancia entre el comportamiento esperado y el comportamiento observado del software creado. La falla es causada por un defecto en algún artefacto.

En la Figura 8-2 se ilustra un ejemplo de la diferencia entre estos tres términos. Al inicio del ejemplo, el analista comete un *error* al escribir un requerimiento difícil de entender, lo que origina un *defecto* en la especificación de requerimientos. El defecto es, en este caso, el requerimiento escrito de manera confusa o incompleta.

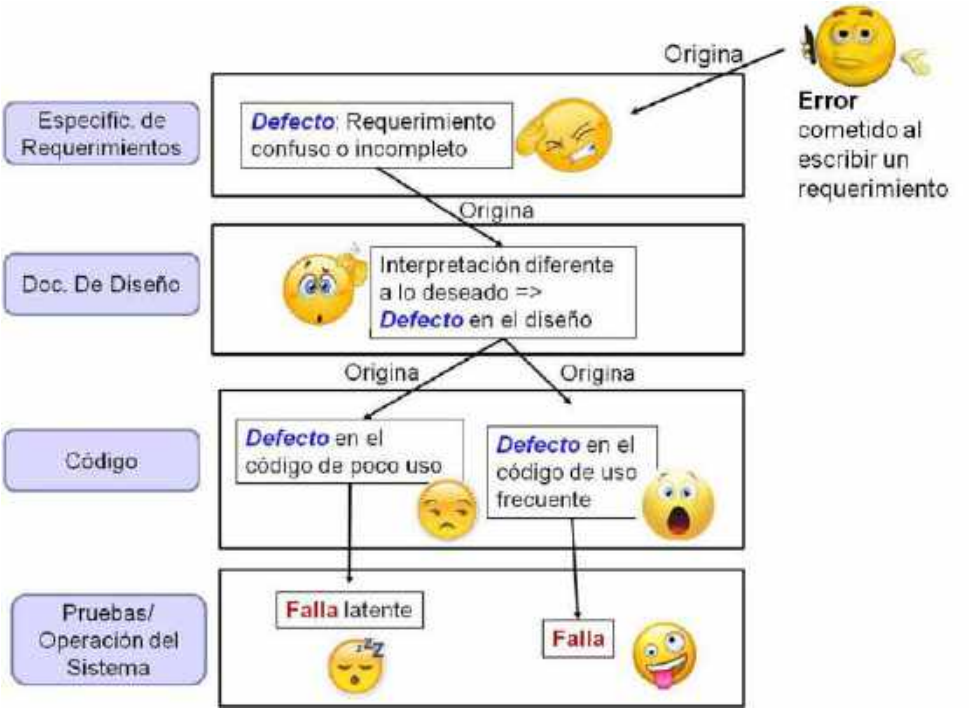


Figura 8-2: Error, defecto y falla

El defecto en la especificación de requerimientos provoca que el diseñador no interprete correctamente lo que se quiso decir al redactar el requerimiento. Este mal entendido origina a su vez un *defecto* en el diseño. El defecto en el diseño puede originar un defecto en un elemento de código que casi no se usa, o un defecto en un elemento de código de uso frecuente. Inclusive, un mismo defecto en el diseño podría originar defectos en varios elementos de código (de uso frecuente y/o de poco uso). Cuando se ejecuta el elemento de código que tiene un defecto entonces se produce una *falla*.

Entonces, el *defecto* está en un artefacto (documento o código) mientras que la *falla* se produce cuando se ejecuta el código.

Se dice que existe una *falla latente* cuando existe un elemento de código que tiene un defecto, pero nadie se ha dado cuenta de su existencia porque este código aún no se ha ejecutado.

8.2.3 Diferencia entre validación y verificación

A continuación, se explica el significado de dos de las actividades que se llevan a cabo durante un proceso de control de calidad.

- **Verificación**.- Se comprueba que el sistema cumple con su especificación de requerimientos.

- **Validación**.- Se determina, en una fase específica del desarrollo del proyecto, si un componente del sistema cumple con las condiciones que se impusieron para ese componente al inicio de la fase. Se comprueba que se satisfacen las expectativas del usuario o del mercado, aunque éstas no hayan sido escritas en la especificación de requerimientos.

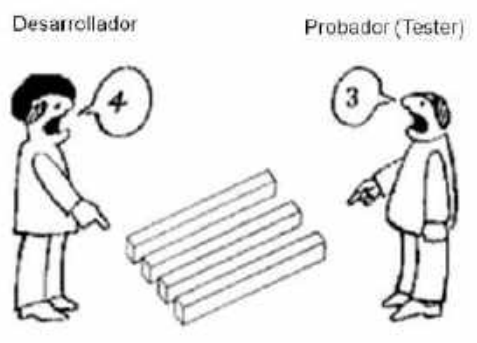
Entonces, mientras que en la *verificación* se comprueba que el sistema cumple con su especificación de requerimientos, en la *validación* se comprueba que se cumplan ciertas condiciones preestablecidas y que se satisfacen las expectativas del usuario.

8.3 Pruebas

La fase de pruebas es indispensable en todo proyecto de software. Ningún producto de software con un mínimo grado de complejidad está exento de defectos que, al menos en potencia, generan fallas. No es viable entregar un producto sin haber pasado por un proceso de prueba que proporcione los niveles de confianza necesarios para poder usar el sistema construido.

Existen diferentes definiciones de prueba. Según el Cuerpo de Conocimiento de la ingeniería de software [softwareEBOK, 2014] "Prueba es la actividad que se lleva a cabo para evaluar la calidad de un producto y mejorarla mediante la identificación de su defectos y problemas".

El estándar ANSI/IEEE 610.12-1990 define prueba como "el proceso de operar un sistema o componente bajo condiciones específicas, observar y registrar los resultados, y realizar una evaluación de algún aspecto del sistema o componente".



[6]Las pruebas del software consisten en un conjunto de actividades encaminadas a identificar defectos, tanto durante el desarrollo del producto software como durante la ejecución del mismo. Éstas deben abarcar cada una de las fases de desarrollo del producto software, y no centrarse solamente en la ejecución del software, ya que en cada una de éstas es posible detectar defectos. Las pruebas del software pueden mostrar la presencia de defectos, pero no pueden garantizar la ausencia de los mismos.

Al ejecutar una prueba se contrasta el comportamiento observado contra el comportamiento esperado del sistema o componente. Es importante hacer notar que el diseño y la ejecución de las pruebas tienen como objetivo la detección de fallas. Si la gran mayoría de pruebas que se construyen no detectan ninguna falla, entonces se habrán desperdiciado los recursos (tiempo y dinero).

8.3.1 El proceso de pruebas

El proceso de pruebas puede verse como parte del proceso de desarrollo de software, o como un proceso aparte. Este proceso, que se ilustra en la Figura 8-3, es una actividad compleja que requiere de una planeación. La planeación se documenta en el *plan de pruebas*, en este documento se especifican las técnicas y métodos a utilizar, así como el ambiente, herramientas necesarias y el calendario previsto.

Durante el *diseño de las pruebas* se identifican los elementos que deben ser probados. Se especifican las condiciones y datos de prueba en función de los elementos a probar. Se diseña la estructura de las pruebas y el ambiente que requieren. Las pruebas se estructuran, se organizan por grupos y categorías, y se les asignan prioridades según un análisis de riesgos.

[6] https://lisaabrego.files.wordpress.com/2015/01/funny-optical-illusions-eye-tricks-11.jpg

La *ejecución de las pruebas* se lleva a cabo siguiendo un *protocolo de pruebas* que consta de un procedimiento de prueba y una lista de casos de prueba por ejecutarse. El resultado de las pruebas se reporta en una *bitácora de pruebas*, conocida como *Test Log.* Y cuando se encuentra una falla se genera un reporte de falla. Durante el diseño y ejecución de las pruebas se lleva a cabo un *control* que tiene como objetivos: monitorear el progreso, medir y analizar los resultados, realizar acciones correctivas y tomar decisiones estratégicas.

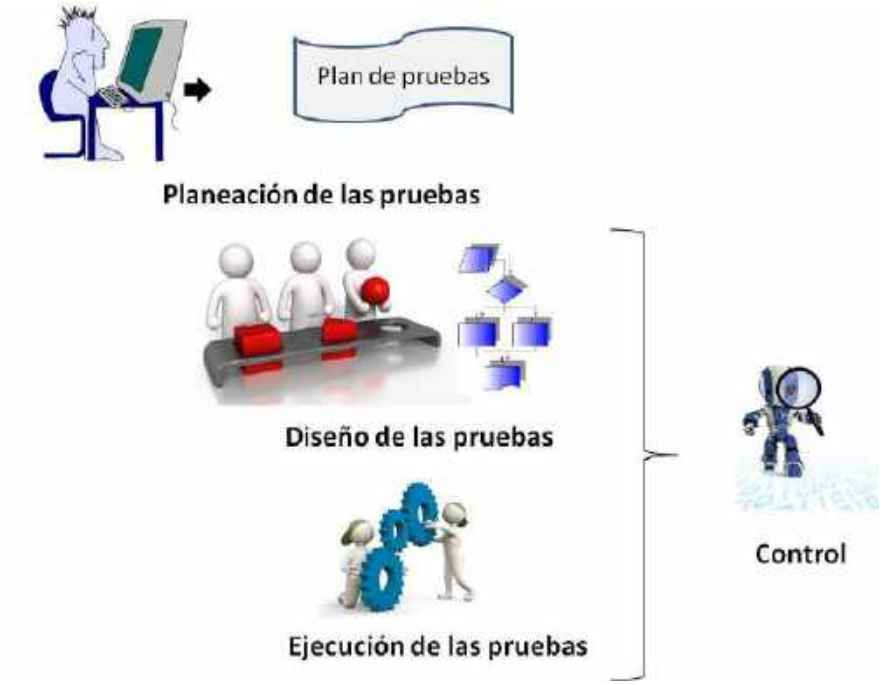


Figura 8-3: El proceso de pruebas

8.3.2 Tipos de prueba

Existen diversas formas de clasificar las pruebas. Whittaker (2002) clasifica las pruebas desde dos puntos de vista: 1) pruebas funcionales y estructurales, también llamadas de caja negra y caja blanca, 2) pruebas unitarias, de integración y de sistema. A continuación, se explican estas dos clasificaciones.

8.3.2.1 Pruebas de caja negra y caja blanca

Las ***pruebas de caja negra*** son aquellas en las que el producto es visto como una caja sin luz en donde no es posible ver la forma en la que éste lleva a cabo sus funciones, lo único visible son sus interacciones con lo exterior, por lo que lo solo se consideran las entradas y las salidas del software que se va a probar. Las pruebas de caja negra, o *funcionales* tienen el punto de vista del usuario porque toman en cuenta los requerimientos, es decir, se prueba que la función requerida se ejecute correctamente, sin importar cómo está implementada.

El propósito de una prueba funcional es buscar discrepancias con la especificación de requerimientos y no demostrar que el programa cumple con ésta cabalmente [Myers, 2004]. Para esto se deben considerar condiciones inválidas y entradas inesperadas al momento de diseñarla.

Las ***pruebas de caja blanca*** son aquellas en donde todos los detalles y la forma en la que funciona el producto están disponibles para el diseñador de pruebas, así que puede diseñar casos de prueba para cada uno de estos detalles. El objetivo de las pruebas de caja blanca es verificar que funcionen diferentes casos que puede haber en la ejecución de un programa. Es importante mencionar que en ocasiones es imposible probar absolutamente todos los casos posibles.

8.3.2.2 Pruebas unitarias, de integración y de sistema

Otra de las formas de clasificar las pruebas es por el objeto que se desea probar [Whittaker, 2002; softwareEBOK, 2014] tal como se ilustra en la Figura 8-4. Las *pruebas unitarias* realizan pruebas en un módulo por separado. Las *pruebas de integración* se realizan para mostrar que, aunque los módulos hayan pasado sus pruebas unitarias, hay problemas al momento de integrar unos con otros. Las *pruebas de sistema* prueban al sistema entero. Por lo general, las fallas de funcionamiento ya se detectaron durante las pruebas unitarias y las de integración. Las *pruebas de sistema* se llevan a cabo para probar los requisitos no funcionales (por ejemplo, seguridad, desempeño, exactitud y confiabilidad) [softwareEBOK, 2014].

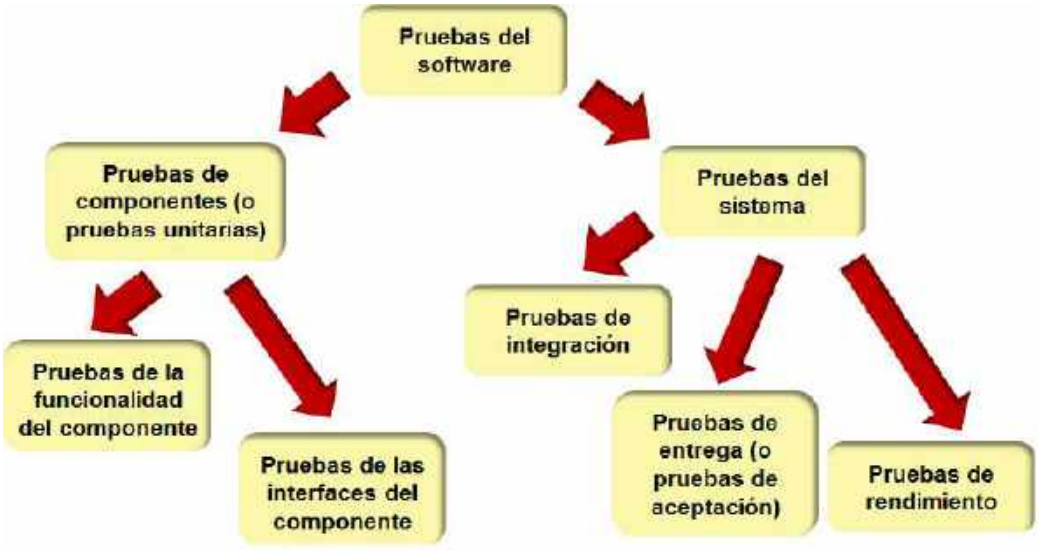


Figura 8-4: Algunos tipos de pruebas del software

La Figura 8-5 ilustra un proceso incremental de pruebas de integración.

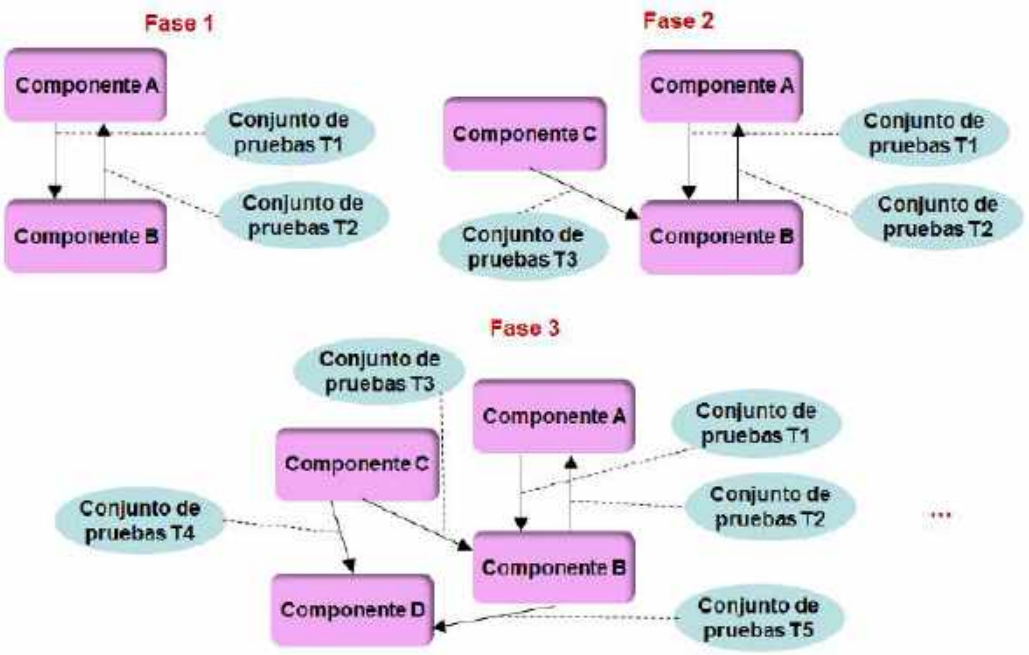


Figura 8-5: Proceso incremental de pruebas de integración

8.3.2.3 Otros tipos de prueba

Existen muchos otros tipos de prueba. Las ***pruebas de aceptación*** son el instrumento para que el cliente acepte y pague el producto. Con las *pruebas de aceptación* se demuestra al cliente que el sistema realmente satisface sus requerimientos. Por lo tanto, una tarea importante es planificar cómo **verificar cada requerimiento**.

Las ***pruebas de regresión*** se llevan a cabo para verificar que no ocurrió una regresión en la calidad del producto luego de un cambio (por ejemplo, al corregir un defecto). Con estas pruebas se asegura que los cambios no introducen un comportamiento no deseado en otras áreas ya probadas. Implican la re-ejecución de alguna o todas las pruebas realizadas anteriormente.

Con las ***pruebas de desempeño*** se miden aspectos tales como: el tiempo de respuesta con tráfico, rendimiento bajo ciertas condiciones de carga de trabajo y configuración, la tasa de atención a peticiones, etc.

Con las ***pruebas de estrés*** se somete el sistema a cargas pesadas de datos o de actividad en un tiempo corto.

Enlistamos también ejemplos de otros tipos de prueba: de seguridad, de usabilidad, de almacenamiento, de configuración, de compatibilidad, de instalación, de confiabilidad, y de recuperación a fallas.

8.4 Gestión de la Configuración

La gestión de la configuración tiene como objetivo controlar los cambios que se generan a lo largo de un proyecto de software para que los elementos que lo componen (código, documentos y datos) estén actualizados y contengan la información correcta. Se identifica la configuración del sistema en distintos puntos del tiempo y con ayuda de herramientas automáticas se controlan los cambios del sistema.

En el desarrollo de sistemas de software, y en particular en el desarrollo de software a gran escala, la gestión de la configuración es en sí una fase transversal del proceso de desarrollo, y tiene una gran importancia en la gestión de la evolución de un sistema de software. Ésta inicia cuando comienza el proceso de desarrollo de software y concluye cuando el software ha sido liberado. Durante la gestión de la configuración del sistema de software se desarrollan y aplican estándares y procedimientos que permitan administrar la evolución, las versiones y los cambios de un sistema de software.

Se dice que en la ingeniería de software *la única constante es el cambio*. La necesidad de un cambio puede originarse por diversas razones, por ejemplo: porque se detectó un defecto, porque que el problema se comprendió mejor conforme se fue avanzando en el desarrollo del proyecto, porque se encontró una mejor solución al problema, o porque las necesidades del cliente cambiaron. El seguimiento y control de los cambios comienza cuando se inicia el proyecto y termina solo cuando el software queda fuera de uso.

Las actividades básicas de la gestión de la configuración son:

1.- *Identificación de los elementos de la configuración del sistema*.- Se proporciona un identificador a cada uno de los elementos del sistema, es decir, a cada módulo software, a cada documento y a los elementos de la base de datos.
2.- *Control de versiones*.- Se establece un sistema para nombrar las versiones del sistema, normalmente se usan números o fechas.
3.- *Control de cambios*.- Es un subproceso en el que se evalúa la necesidad y el costo de cada cambio. En caso de que el cambio se apruebe se genera la orden de cambio, y se le da seguimiento a su implantación, revisión y a la nueva versión que éste origina.

Con una correcta gestión de la configuración se garantiza que todos los participantes en el desarrollo del proyecto disponen de la versión adecuada de los productos que la empresa maneja. Además, se facilita el mantenimiento del sistema porque proporciona la información precisa para valorar el impacto de los cambios.

8.4.1. Control de cambios

La principal fuente de cambios, tanto durante el desarrollo de un sistema de software como durante el tiempo de vida de dicho sistema, viene dada por el cambio de los requerimientos funcionales. Los requerimientos funcionales de un sistema software pueden cambiar según las nuevas expectativas y necesidades de los clientes. Cada cambio se analiza para determinar si procede y, de ser el caso, el cambio debe llevarse a cabo de forma organizada y correcta. Cuando el cambio se deriva de los requerimientos del sistema de software, entonces comúnmente el cliente debe trabajar con el equipo encargado de la gestión de la configuración del sistema, para evaluar el impacto del cambio y decidir su realización. Otra fuente de cambios viene dada por fallas y defectos descubiertos durante la fase de pruebas del sistema. El control de cambios se basa en el uso de procedimientos humanos y herramientas automáticas, que proporcionan un mecanismo para la evaluación del cambio solicitado, y de ser el caso, la realización, seguimiento y revisión del mismo. Los procedimientos del control de cambios incluyen el análisis de costos y los beneficios de los cambios propuestos para determinar si merecen la pena llevarse a cabo. Una vez que se ha aceptado un cambio, se identifican los componentes y elementos de configuración del sistema de software que deben cambiar. El control de cambios, como parte de la gestión de la configuración del sistema software, abarca las siguientes actividades (Sommerville, 2015):

- Identificación del cambio.
- Control del cambio.
- Implementación del cambio.
- Información del cambio a todas las partes afectadas.

La Figura 8-6 muestra la secuencia de las principales actividades involucradas en el control de cambios.

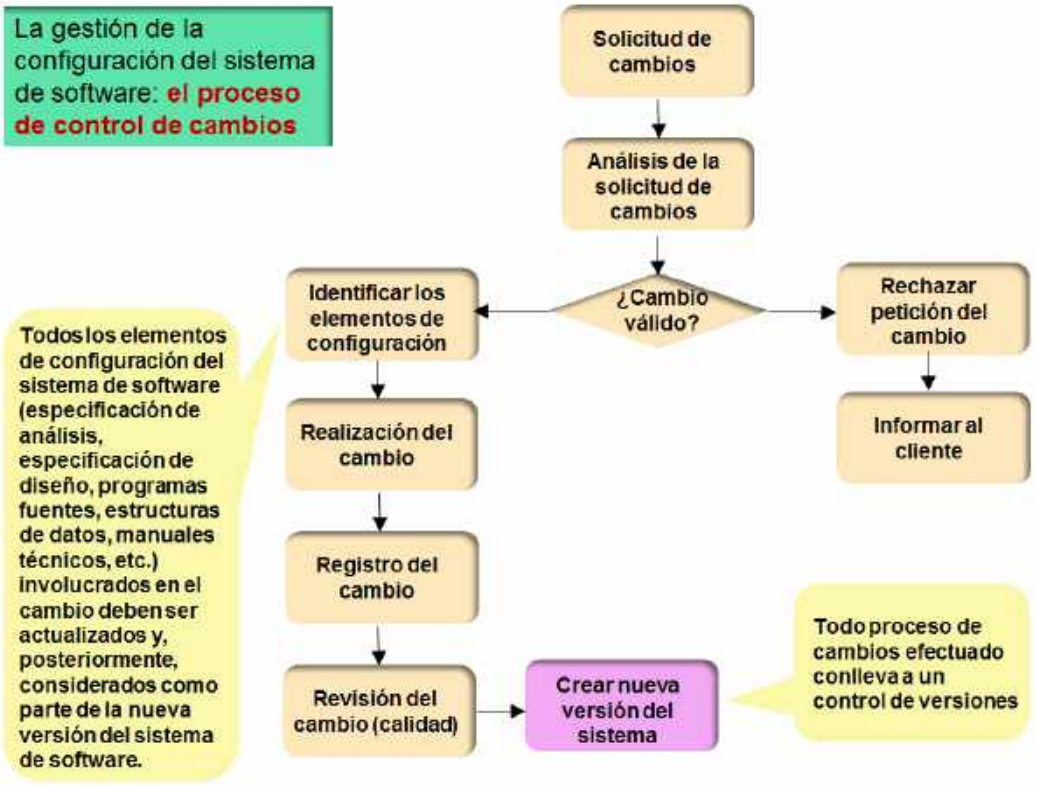


Figura 8-6: Principales actividades de la gestión de cambios

8.4.2. Control de versiones

El control de versiones se refiere al uso de procedimientos y herramientas para gestionar las versiones de los elementos de configuración del sistema de software tales como: la especificación de los requerimientos, la especificación del diseño arquitectónico, la especificación del diseño detallado, las estructuras de datos, los programas fuentes, etc. Como ya habíamos mencionado, todo control de cambios conlleva a un control de versiones.

El control de versiones implica la identificación y al mantenimiento de los registros de las diferentes versiones del sistema de software. Se toman en cuenta los siguientes aspectos:

- Que las diferentes versiones del sistema se puedan recuperar cuando éstas sean requeridas.
- Que estas versiones no se cambien de forma accidental o errónea por parte del equipo de desarrollo.

Una ***versión*** de un sistema de software es una instancia o copia del sistema, que difiere de alguna forma de otras descripciones. La diferencia entre versiones de un sistema de software se debe a alguno de los siguientes aspectos:

- Cambios en la funcionalidad o requerimientos.
- Cambios en las estructuras de datos.
- Cambios en el diseño.
- Cambios en las interfaces gráficas de usuario.
- Cambios en la configuración de hardware.

Para crear una versión particular de un sistema de software, es necesario especificar la versión de cada uno de los componentes que debe formar parte de esta versión del sistema. Entre las principales técnicas utilizadas en la identificación de componentes se encuentran las siguientes:

- **Numeración de versiones**. Cuando se usa la técnica de numeración de versiones, a cada componente se le asigna un número de versión explícito y único. Ésta es la técnica de identificación de componentes más utilizada.

- **Identificación basada en atributos**. Esta técnica se basa en identificar las versiones considerando un conjunto de atributos de identificación y valores para dichos atributos. Los principales atributos que permiten identificar las versiones son los siguientes: el lenguaje de desarrollo, estado o fase de desarrollo, plataforma de hardware, tecnología predominante y fecha de creación, entre otros atributos.

- **Identificación orientada al cambio**. Esta técnica está mucho más orientada a la identificación de la versión del sistema de software de forma global, que a la identificación individual de los componentes que integran el sistema. Cada nueva versión del sistema de software tiene asociado un conjunto de cambios, el cual describe los cambios efectuados a los componentes del sistema, que fueron requeridos para implementar el cambio que encierra la nueva versión.

La Figura 8-7 ilustra la estructura de derivación de versiones resultante, al utilizar la técnica numeración de versiones en el sistema computacional Evolution, una plataforma bioinformática para la simulación del plegamiento de secuencias de proteínas.

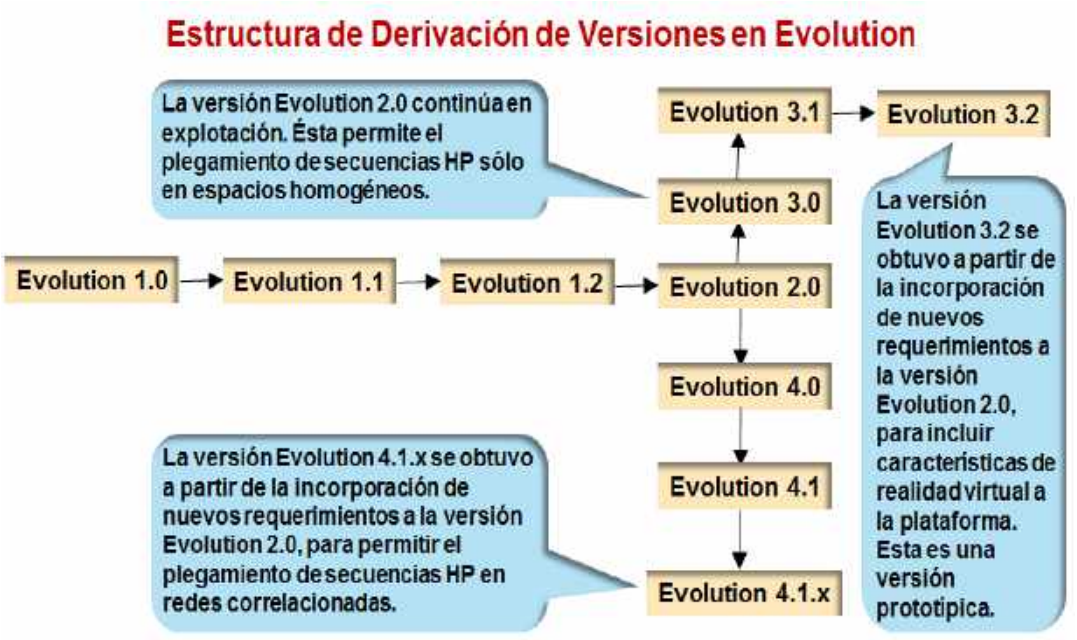


Figura 8-7: Ejemplo de control de versiones

8.5 Soporte técnico y Mantenimiento

8.5.1. Soporte técnico

Soporte técnico es el servicio que se proporciona al cliente cuando el sistema ya está en operación con el objetivo de solucionar los problemas que se le presentan. Incluye la capacitación del usuario para que pueda trabajar con el sistema, la solución de dudas y, cuando es necesario, corregir las fallas detectadas en el sistema.

El soporte técnico se refiere a la asistencia que se proporciona a los usuarios, clientes o propietarios de un producto de software o un producto de hardware, ante los problemas o dificultades que puedan surgir durante su uso. La asistencia de soporte técnico se puede llevar a cabo por varias vías, entre las que se encuentran:

- Vía telefónica.
- Presencial (en sitio).
- Correo electrónico.
- Chat.
- software de aplicación.
- Asistencia técnica remota.

Con el impetuoso auge de las tecnologías de información y comunicaciones que ha tomado lugar en los últimos años, técnicas tales como el correo electrónico, el chat, el software de aplicación y la asistencia técnica remota, juegan un importante papel en la asistencia de soporte técnico. En los últimos años los *help desk*, también conocidos como mesas de ayuda o centro de atención al usuario, han constituido una de las mejores alternativas para la asistencia del soporte técnico. Los *help desk* integran el software de aplicación con la asistencia técnica remota, a través de un conjunto de recursos tecnológicos y humanos, para proporcionar soluciones efectivas al usuario, ante los problemas que puedan surgir ante el uso de un determinado producto de software o hardware.

8.5.2. Mantenimiento

El *mantenimiento* es el conjunto de actividades que se llevan a cabo para hacer actualizaciones en un sistema de software cuando ya está en operación. El mantenimiento del software es una fase posterior al desarrollo y liberación del producto software y está encaminada principalmente a:

- Incorporar nueva funcionalidad al producto software.
- Mejorar la funcionalidad presente en el producto software.
- Mejorar el rendimiento del producto software.
- Corregir defectos no detectados durante la fase de pruebas del software.

Un sistema de software desarrollado atendiendo a los principios de reusabilidad y extensibilidad, garantizará un proceso de mantenimiento flexible. Comúnmente, el mantenimiento del software está orientado a la incorporación de mejoras a la funcionalidad del sistema software, y no tanto a la corrección de errores.

Sommerville (2007) clasifica las actividades de mantenimiento en tres tipos:

1.- *Mantenimiento para reparar defectos del software*.- También se le conoce como *mantenimiento correctivo*. Arreglar el defecto puede implicar cambios en código solamente, o incluso cambios en el diseño.

2.- *Mantenimiento para adaptar el software a diferentes ambientes de operación*.

3.- *Mantenimiento para añadir o modificar funcionalidad al sistema*.- Se realiza cuando hay cambios en los requerimientos cuyo impacto en el sistema no es demasiado grande.

Una tarea primordial que debe preceder a las actividades propias de la gestión del mantenimiento es la predicción del mantenimiento. La predicción del mantenimiento implica anticipar aspectos clave [Sommerville, 2012], tales como:

- ¿Qué partes del sistema son más probables de afectarse por las peticiones de cambio?
- ¿Cuántas peticiones de cambio se esperan?
- ¿Qué partes del sistema serán más costosas de mantener?
- ¿Cuáles serán los costos de mantenimiento durante el período de vida del sistema?
- ¿Cuáles serán los costos de mantenimiento del sistema en el próximo año?

El proceso de mantenimiento del software involucra diferentes actividades, tanto a nivel de la organización propietaria o usuaria del sistema de software, como a nivel del equipo de desarrollo que se encargará del mantenimiento del sistema de software. En [Pressman, 2010] se identifican las siguientes actividades:

- Organización del mantenimiento.
- Informe o petición del mantenimiento.
- Flujo de sucesos resultante del informe o petición del mantenimiento.
- Registro de información del mantenimiento.
- Evaluación de las actividades de mantenimiento de software.

La *curva de Rayleigh* (ver Figura 8-8) es una estimación del número de fallas que presenta un sistema de software a lo largo del tiempo una vez que éste entra en operación. Se espera que la mayoría de los problemas surjan al principio de la entrega y que posteriormente éstos disminuyan. Además, mientras más se use un sistema al inicio de su entrega, los problemas se detectarán más rápidamente.

Figura 8-8: La curva de Rayleigh

Los proyectos exitosos normalmente siguen en el mercado. Se liberan nuevas versiones y se le hacen actualizaciones, por lo que el mantenimiento es una actividad que normalmente dura más tiempo que el desarrollo del proyecto.

8.6 Resumen

A continuación, presentamos un resumen con las ideas principales que se abordaron en este capítulo:

El *defecto* está en un artefacto (documento o código) mientras que la *falla* se produce cuando se ejecuta el código.

En la *verificación* se comprueba que el sistema cumple con su especificación de requerimientos, en la *validación* se comprueba que se cumplan ciertas condiciones preestablecidas y que se satisfacen las expectativas del usuario.

En las *pruebas de caja blanca* se prueba que la función requerida se ejecute correctamente, sin importar cómo está implementada. Mientras que en las *pruebas de caja negra* se hacen para verificar que funcionen diferentes casos que puede haber en la ejecución de un programa, por lo tanto, se sabe cómo están implementadas las funciones en el código.

El soporte técnico incluye la capacitación del usuario, la solución de dudas y el mantenimiento. El mantenimiento consiste en la corrección de fallas que surgen durante la operación.

8.7 Cuestionario

1.- ¿Qué es calidad del software?

2.- Indica los artefactos que se evalúan en cada una de las siguientes fases del desarrollo de un proyecto de software.

- Durante el *Inicio del proyecto*:

- Durante el *Análisis y Especificación de Requerimientos*:

- Durante el *diseño*:

- Durante la *codificación*:

- Durante las *pruebas unitarias* (de módulo):

- Durante las *pruebas del sistema*:

- Durante la *gestión de la configuración*:

3.- ¿Cuáles son las tres actividades principales del control de calidad del software?

4.- ¿Cuál es la diferencia entre *defecto* y *falla*?

5.- ¿Cuál es la diferencia entre *verificación* y *validación*?

6.- ¿Cuáles son las actividades del proceso de pruebas?

7.- ¿Cuál es la diferencia entre las pruebas de caja negra y las de caja blanca?

8.- ¿En qué consiste la gestión de la configuración?

9.- ¿Cuál es la diferencia entre soporte técnico y el mantenimiento?

9 Modelos de Desarrollo en Cascada

9.1 Objetivo general del capítulo

- Conocer las fases fundamentales del proceso de desarrollo en cascada identificando el artefacto que se requiere a la entrada y a la salida de cada una de estas fases.
- Conocer las características y limitaciones del modelo en cascada.
- Conocer los principales modelos que se derivan del modelo en cascada.
- Conocer los roles de los ingenieros de software en las diferentes etapas del proceso de desarrollo.

9.2 Introducción

El desarrollo de software es una *secuencia de pasos* que transforman un modelo en una implementación. El modelo en cascada es un proceso de desarrollo de software al que también se le llama "lineal" o "secuencial". Éste propone fases separadas que se llevan a cabo de manera secuencial, es decir, no se empieza con la siguiente hasta que se haya terminado la anterior.

El modelo de cascada (también conocido como cascada pura) es el predecesor de todos los modelos de ciclo de vida y ha servido de base para otros. En un modelo en cascada, un proyecto progresa a través de una secuencia ordenada de etapas, partiendo desde el concepto inicial del software hasta la prueba del sistema. El proyecto realiza una revisión al final de cada etapa para determinar si está preparado para pasar a la siguiente etapa.

El modelo de cascada se utiliza correctamente para ciclos de productos en los que se tiene una definición estable del producto, y también cuando se está trabajando con metodologías técnicas conocidas. Puede constituir una elección correcta para el desarrollo rápido cuando se está construyendo una versión de mantenimiento bien definida de un producto existente o migrando un producto existente a una nueva plataforma. Ayuda a minimizar los gastos de la planeación porque permite realizarla sin problemas.

Como ilustra la Figura 9-1, el modelo en cascada consta de 5 etapas, que son las actividades fundamentales en cualquier desarrollo de software.

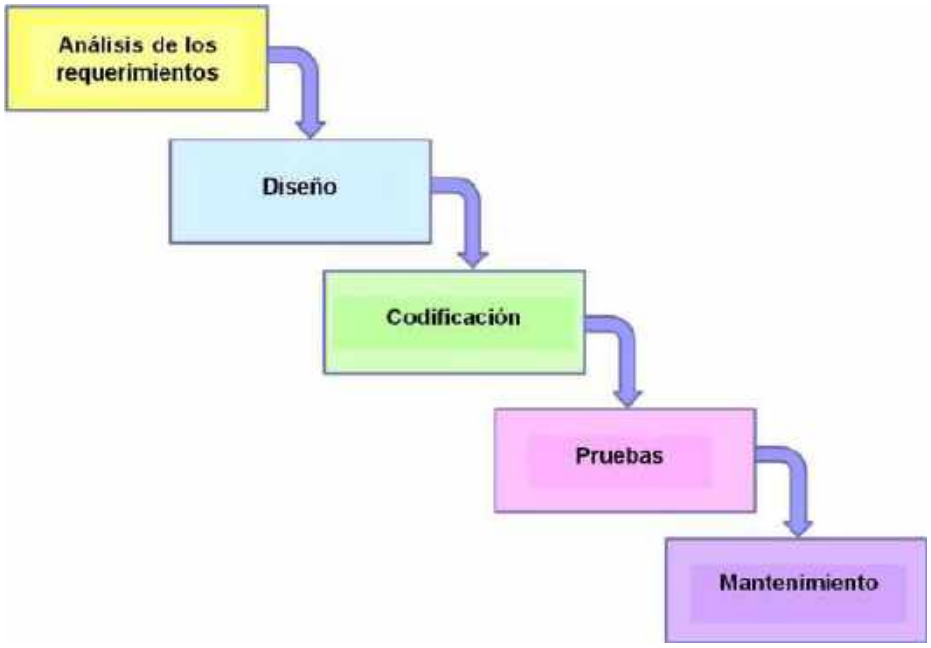


Figura 9-1: El modelo en cascada

Estas etapas son las que hemos estudiado en los capítulos anteriores. A continuación, resumimos sus principales características e ilustramos el artefacto que se requiere a la entrada y a la salida de cada una de las fases.

1.- *Análisis y especificación de requerimientos*.-

Se identifican cuáles son los requerimientos del cliente en base a sus necesidades. Se definen los servicios, metas y restricciones del sistema a partir de consultas con los clientes y usuarios. Los requerimientos se documentan en la **Especificación de Requerimientos.** La entrada y salida de esta etapa se ilustra en la Figura 9-2.

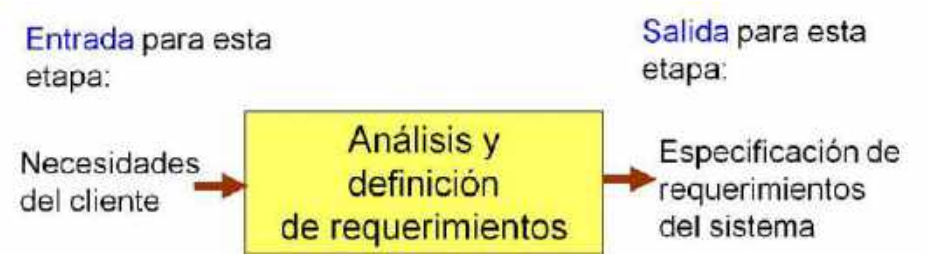


Figura 9-2: Entrada y salida en la fase de análisis y especificación de requerimientos

2.- *Diseño*.-

Durante el diseño de software se determina la estructura del sistema de software y de sus datos antes de iniciar su codificación. El diseño del sistema se divide en dos etapas:

- **Diseño de alto nivel** (Top Level Design: TLD).- El sistema se divide en partes y se establecen las relaciones entre ellas. Se define qué hará cada una de las partes.

- **Diseño detallado** (Detailed Design: DD).- Se establece cómo debe funcionar cada una de las partes del sistema.

La entrada y salida de esta etapa se ilustran en la Figura 9-3.

Figura 9-3: Entrada y salida en la fase de diseño

3.- *Codificación*.-

Durante esta etapa se implementa en código lo que esta descrito en el documento de diseño. Se producen programas ejecutables libres de errores de compilación. La entrada y salida de esta etapa se ilustra en la Figura 9-4.

Figura 9-4: Entrada y salida en la fase de codificación

4.- *Pruebas*.-

El sistema de software es un conjunto de módulos. Primero se prueba cada módulo de forma independiente, es decir, se realizan las *pruebas unitarias*. Después se llevan a cabo las *pruebas de integración y de sistema*, en las que se integra y prueba todos los módulos como un sistema completo para asegurar que se cumplan los requerimientos del software. Después de pasar las pruebas, el sistema de software se entrega al cliente. La entrada y salida de esta etapa se ilustra en la Figura 9-5.

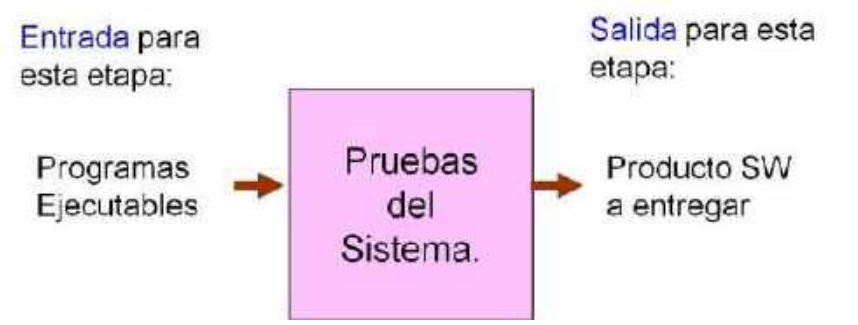


Figura 9-5: Entrada y salida en la fase de pruebas

5.- *Mantenimiento*.-

Una vez que se entrega al cliente, el sistema se instala y se pone en operación. El mantenimiento implica corregir defectos no descubiertos en las etapas anteriores del ciclo de vida y mejorar la implantación de las unidades del sistema. Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida. La tabla de la Figura 9-6 es un concentrado de los artefactos de entrada y de salida para cada una de las fases del modelo en cascada.

Fase	Artefacto de entrada	Artefacto de salida
Requerimientos	Necesidades del cliente	Especificación de Requerimientos
Diseño	Especificación de Requerimientos	Documento de diseño
Codificación	Documento de diseño	Programa ejecutable
Pruebas	Programas ejecutables	Producto software a entregar

Figura 9-6: Artefactos de entrada y salida en las fases del modelo en Cascada

9.3 Características del modelo en cascada y sus limitaciones

El resultado de cada fase es uno o más documentos, los cuales deben ser aprobados después de pasar por algún mecanismo de control de calidad. En teoría, la siguiente fase no debe empezar hasta que la fase previa haya finalizado. Sin embargo, en la práctica, las etapas se superponen y proporcionan información las unas a las otras. Por ejemplo, durante el diseño se identifican problemas con los requerimientos, durante la elaboración de código se encuentran problemas en el diseño y así sucesivamente.

Uno de los principales problemas del modelo en cascada reside en que no trata al software como un proceso de resolución de problemas. El modelo en cascada se derivó del mundo del hardware y presenta una visión de manufactura sobre el desarrollo del software. Pero la manufactura produce un artículo en particular y lo reproduce muchas veces. En cambio, el software no se fabrica de la misma forma, sino que evoluciona a medida que el problema se comprende. Entonces el desarrollo de un sistema de software es más bien un proceso creativo, en el que se desarrollan y evalúan prototipos, se valora la factibilidad de los requerimientos, se compara varios diseños, se aprende a partir de los errores, y finalmente se construye una solución satisfactoria para el usuario.

Otro problema de este modelo es que, una vez finalizada una etapa, no se contempla regresar a ella, y es muy costoso cambiar los requerimientos del cliente, pues cuando regresamos a los requerimientos hay que re-hacer las etapas posteriores. Por lo tanto, el modelo en cascada solo se debe utilizar cuando los requerimientos se comprenden bien y sea improbable que cambien radicalmente durante el desarrollo del sistema. Sin embargo, este modelo de desarrollo es muy importante porque define las actividades que se siguen en la mayoría de los procesos de software.

9.4 Modificaciones al modelo en cascada

Con el fin de solucionar las limitaciones del modelo en cascada surgieron varios modelos que lo modifican de alguna manera. En esta sección presentamos los principales modelos que surgieron a partir de hacer alguna modificación al desarrollo en cascada.

9.4.1 Cascada en V

En la cascada en V cada fase del desarrollo tiene una fase correspondiente de verificación y validación (ver Figura 9-7).

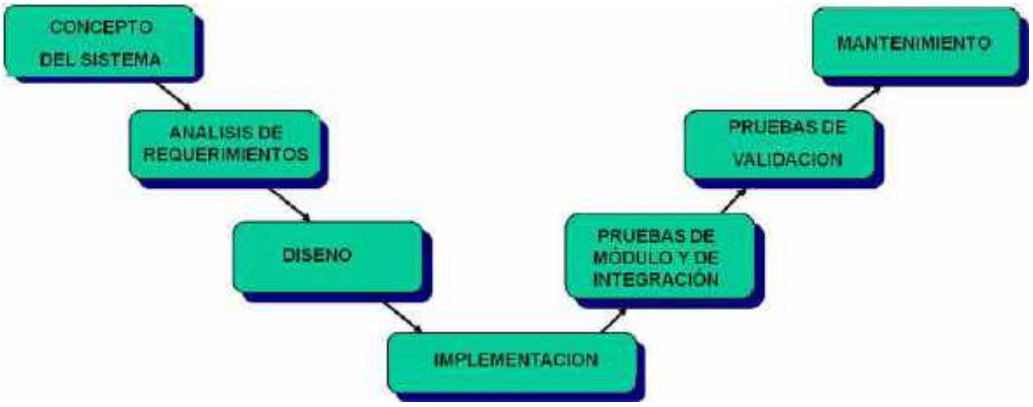


Figura 9-7: Cascada en V

En la Tabla 9-1 se ilustran las ventajas y desventajas del modelo *Cascada en V* según [McConnell, 1997].

Ventajas	Desventajas
■ Produce un sistema robusto y fiable. ■ Evita el flujo descendente de los defectos. ■ Funciona bien para pequeños proyectos donde los requisitos son fáciles de entender.	■ No funciona si los requerimientos no son claros desde el principio. ■ Escribir pruebas por adelantado hace rígido el proceso e impide a los probadores seleccionar la forma más eficaz y eficiente de planificar y ejecutar las pruebas.

Tabla 9-1: Ventajas y desventajas del modelo Cascada en V

9.4.2 Cascada con fases solapadas

En la cascada con fases solapadas se permite avanzar a la siguiente etapa antes de terminar la actual (ver Figura 9-8). El modelo de cascada con fases solapadas puede evitar algunos inconvenientes del modelo de cascada pura, al solapar sus etapas. Es decir, este modelo sugiere un mayor grado de solapamiento. Por ejemplo, sugiere que se debería tener bien hecho el diseño global y quizás a medio hacer el diseño detallado antes de considerar completo el análisis de requerimientos. Otra característica de interés del modelo de Cascada con Fases Solapadas es que se puede reducir sustancialmente la documentación necesaria entre etapas.

Figura 9-8: Cascada con fases solapadas

En la Tabla 9-2 se ilustran las ventajas y desventajas del modelo *Cascada con fases solapadas,* según [McConnell, 1997].

Ventajas	Desventajas
■ La siguiente etapa retroalimenta a la anterior y permite mejorarla. ■ Se pueden descubrir ideas importantes al avanzar en los ciclos de desarrollo. ■ Funciona en proyectos grandes.	■ No funciona si los requerimientos no son claros desde el principio. ■ La realización de actividades en paralelo puede provocar una mala comunicación e ineficacia. ■ Los pasos son más ambiguos, y esto hace más difícil el seguimiento del proceso.

Tabla 9-2: Ventajas y desventajas del modelo Cascada con fases solapadas

9.4.3 Cascada con subproyectos

Otro problema presente en el modelo de cascada pura, desde el punto de vista del desarrollo ágil, es que se supone que se ha terminado completamente una etapa antes de iniciar la siguiente (por ejemplo, no es posible iniciar el diseño detallado si no se ha terminado completamente el diseño global). En la cascada con subproyectos se permite la ejecución de algunas de las tareas de la cascada en paralelo (subproyectos), siempre que se haya realizado una cuidadosa planificación (ver Figura 9-9).

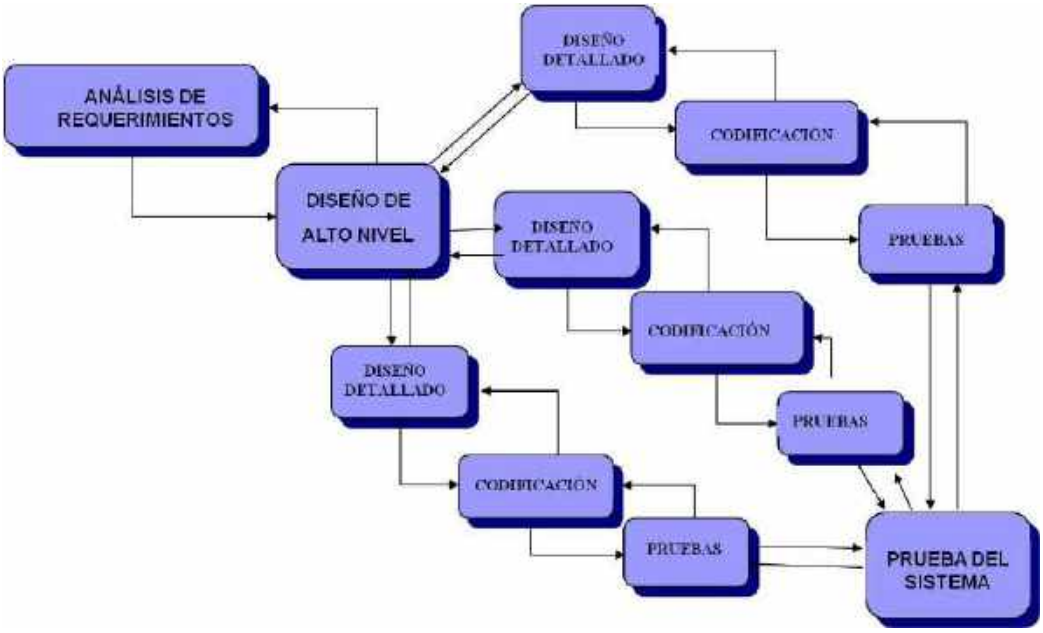


Figura 9-9: Cascada con subproyectos

En la Tabla 9-3 se ilustran las ventajas y desventajas del modelo *Cascada con subproyectos*, según [McConnell, 1997].

Ventajas	Desventajas
■ Cada subsistema lógicamente independiente es un subproyecto que avanza a su propio ritmo. ■ Se produce una buena documentación completa del sistema.	■ Pueden existir interdependencias no previstas entre los subproyectos. ■ No tiene buena aceptación de cambios en los requerimientos.

Tabla 9-3: Ventajas y desventajas del modelo Cascada con subproyectos

9.4.4 Cascada con reducción de riesgos

Otra de las limitaciones del modelo de cascada pura es que requiere la definición completa de los requerimientos antes de comenzar la etapa de diseño global. El modelo de cascada con reducción de riesgos incorpora una espiral en lo alto de la cascada para controlar el riesgo de los requerimientos. A este nivel es posible desarrollar un prototipo de interfaz de usuario, utilizar apuntes, tener entrevistas con los usuarios, observar cómo los usuarios interactúan con algún sistema más antiguo, o utilizar otros métodos que se consideren apropiados para la identificación de los requerimientos (ver Figura 9-10).

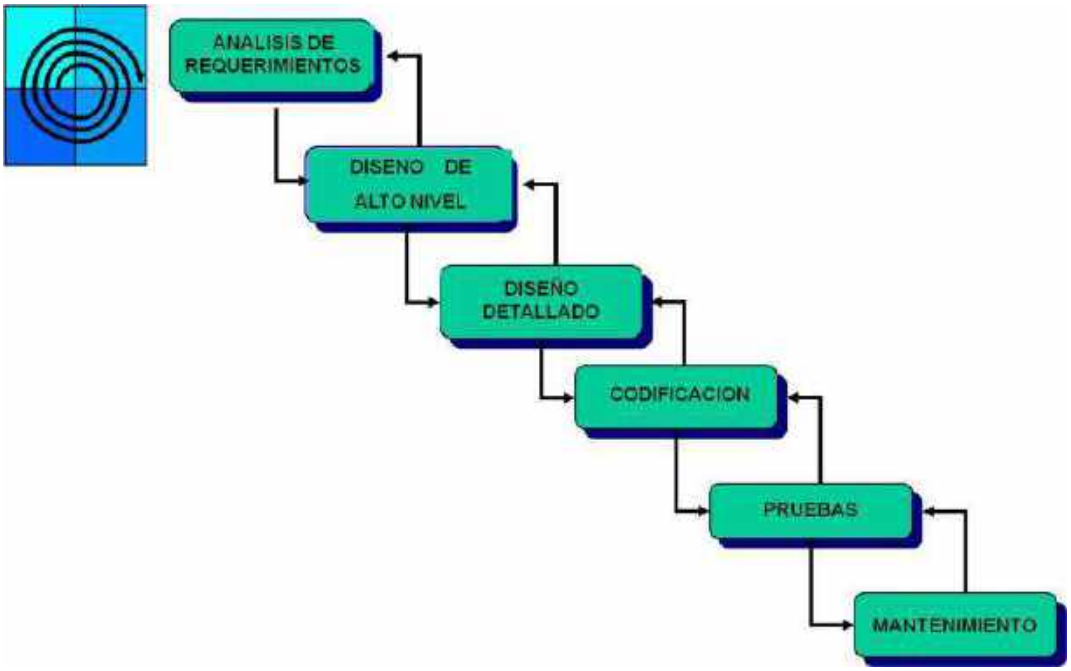


Figura 9-10: Cascada con reducción de riesgos

En la Tabla 9-4 se ilustran las ventajas y desventajas del modelo *Cascada con reducción de riesgos*, según [McConnell, 1997].

Ventajas	Desventajas
■ Se reduce el riesgo de producir un sistema que no cumple con los requerimientos. ■ Se produce una buena documentación completa del sistema.	■ Las modificaciones en los requerimientos afectan bastante una vez que ya se esta trabajando en etapas avanzadas.

Tabla 9-4: Ventajas y desventajas del modelo Cascada con reducción de riesgos

En la Figura 9-11 se hace un resumen de las principales características de los modelos de desarrollo de software derivados del modelo en cascada.

Modelo	Características
Cascada en V	Cada fase del desarrollo tiene una fase correspondiente de verificación y validación
Cascada con fases solapadas	Permite avanzar a la siguiente etapa antes de terminar la actual.
Cascada con subproyectos	Permite la ejecución de algunas de las tareas de la cascada en paralelo (subproyectos), siempre que se haya realizado una cuidadosa planificación.
Cascada con reducción de riesgos	Incorpora una espiral en lo alto de la cascada para controlar el riesgo de los requerimientos: se hacen prototipos de interfaz de usuario, entrevistas, etc.

Figura 9-11: Características principales de los diferentes modelos derivados del Cascada

9.4.5 Los roles de los ingenieros de software en las diferentes etapas del proceso de desarrollo

Cuando los sistemas que se desarrollan en una empresa son grandes, el trabajo se divide por equipos. Así que es necesario que el ingeniero de software sepa comunicar bien sus ideas y escuchar las de los demás. En la Figura 9-12 se ilustran los diferentes roles que puede tener un ingeniero de software en base a su experiencia. Como el análisis y especificación de requerimientos es la fase más importante, un error en esta etapa es el más costoso, por lo que se nombra como analista a las personas con más experiencia.

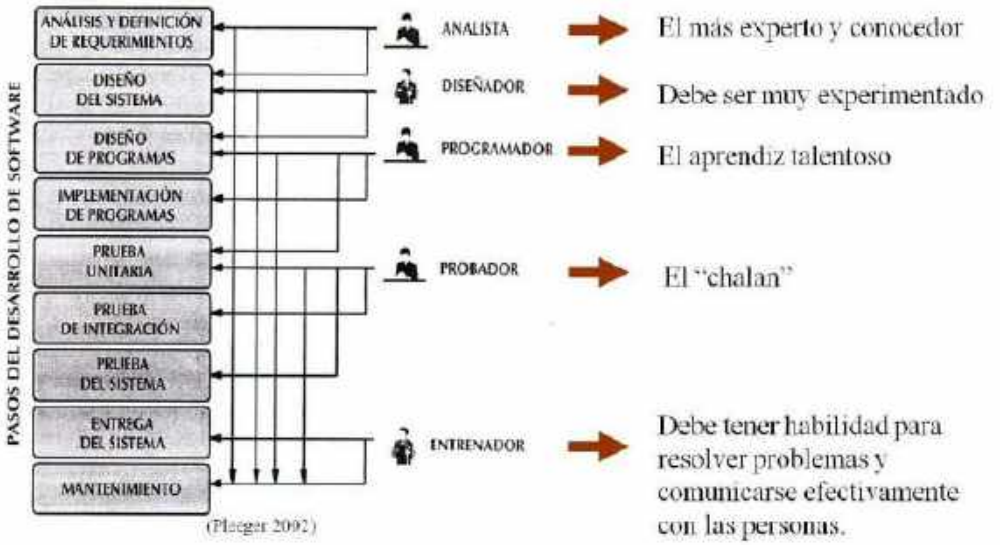


Figura 9-12: Los diferentes roles de los ingenieros de software (adaptado de Pfleeger 2002)

Siguiendo el orden de experiencia e importancia, un diseñador juega también un papel clave, por lo que se requiere que tenga también mucha experiencia. El programador debe ser talentoso y confiable, cuando adquiere suficiente experiencia pasa a ser diseñador. Los principiantes entran normalmente como probadores (*testers*) o en mantenimiento y soporte técnico, ya que son buenas actividades para familiarizarse con el sistema que maneja la empresa y si cometen un error no es tan grave como lo sería en las etapas previas.

La Figura 9-13 ofrece un panorama más amplio, incluyendo otros roles y responsabilidades, requeridos durante el desarrollo de software a gran escala.

Figura 9-13: Diferentes roles en el desarrollo de software a gran escala

Finalmente, la Figura 9-14 ilustra dos alternativas comúnmente utilizadas en la conformación de los equipos de desarrollo de software, cuando se trata de abordar proyectos de software a gran escala.

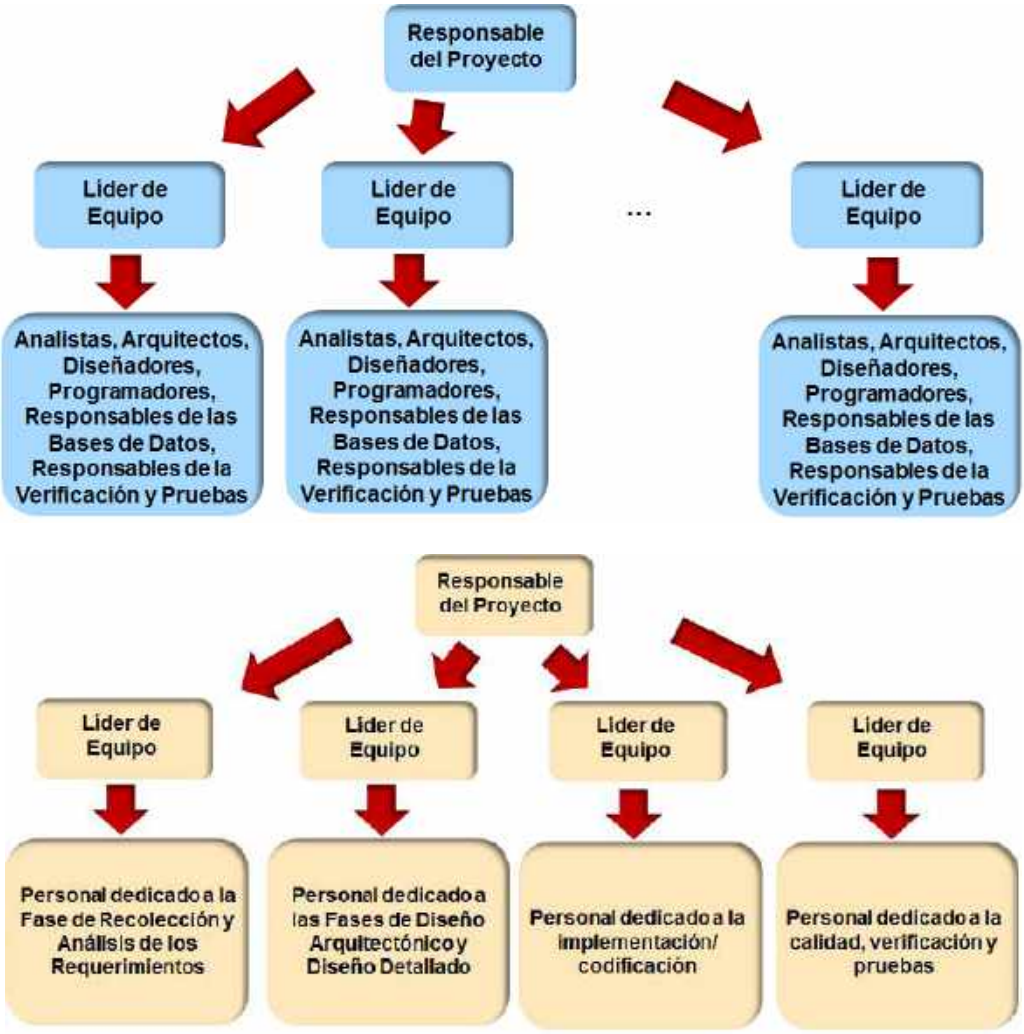


Figura 9-14: Configuración de los equipos para desarrollo de software a gran escala

9.5 Resumen

El modelo en cascada es un proceso de desarrollo de software al que también se le llama “lineal” o “secuencial”. Éste propone fases separadas que se llevan a cabo de manera secuencial, es decir, no se empieza con la siguiente hasta que se haya terminado la anterior. El modelo en cascada consta de 5 etapas que son: análisis de los requerimientos, diseño, codificación, pruebas y mantenimiento.

Como el modelo en cascada es muy inflexible, existen variaciones que pretenden mejorar este modelo, como son: cascada en V, cascada con fases solapadas, cascada con subproyectos y cascada con reducción de riesgos.

9.6 Cuestionario

1.- Enlista los artefactos de entrada y salida para cada una de las fases del modelo en cascada que se indican en la siguiente tabla:

Fase	Artefacto de entrada	Artefacto de salida
Requerimientos		
Diseño		
Codificación		
Pruebas		

2.- ¿Por qué el modelo en cascada solo se debe utilizar cuando los requerimientos se comprenden bien y sea improbable que cambien radicalmente?

3.- Explica las principales características del modelo de desarrollo Cascada en V.

4.- Explica las principales características del modelo de desarrollo Cascada con fases solapadas.

5.- Explica las principales características del modelo de desarrollo Cascada con subproyectos.

6.- Explica las principales características del modelo de desarrollo Cascada con reducción de riesgos.

10 Modelos de desarrollo Evolutivos

10.1 Objetivos específicos del capítulo

- Conocer los principios del proceso del desarrollo evolutivo.
- Conocer los principales modelos evolutivos.
- Conocer las características y limitaciones de los modelos evolutivos.

10.2 Introducción

[7]El desarrollo evolutivo consiste en construir el sistema poco a poco en lugar de hacerlo todo "de un jalón", como en el modelo en cascada. El sistema evoluciona en cada etapa porque se le van agregando nuevas características. Los modelos evolutivos tienen como objetivo principal reducir el riesgo y la incertidumbre en el desarrollo. Se utilizan cuando se pueden entregar versiones preliminares con los requerimientos más urgentes o con las partes del sistema que se comprenden mejor. También se utilizan cuando se requiere explorar los requerimientos, porque no se tienen bien definidos. El desarrollo empieza con las partes del sistema que se comprenden mejor y se le van agregando al sistema nuevas características.

En los *modelos evolutivos*, los requerimientos se trabajan al inicio de cada iteración para aumentarlos, corregirlos o redefinirlos.

[8]Con los modelos evolutivos se van mostrando al cliente versiones parciales las cuales ***permiten obtener retroalimentación*** que ayuda a mejorar el sistema. De esta manera se va construyendo el sistema que el cliente requiere y se evitan los problemas que se presentan cuando se integra un código muy grande, pues el sistema se va integrando desde la primera entrega parcial. Para que este modelo sea útil se debe poder comenzar con algunos requerimientos prioritarios y dejar para los ciclos posteriores los demás requerimientos, además hay que contar con la participación del usuario, quien debe dedicar tiempo a evaluar y retroalimentar las entregas parciales [Braude, 2007].

[7] https://dumielauxepices.net/sites/default/files/software-development-clipart-project-progress-771504-9019776.png

[8] https://dahianasanchez.files.wordpress.com/2013/09/imagen-asertiva-2.jpg

10.3 Tipos de modelos evolutivos

Figura 10-1: Tipos de modelos evolutivos según Sommerville (2007)

En la Figura 10-1 se muestran los diferentes tipos de modelos evolutivos. Según Sommerville (2007) los dos tipos principales de modelo evolutivo son el desarrollo exploratorio y los prototipos desechables. Además, existen cuatro tipos de desarrollo exploratorio que son, el incremental, el iterativo, el prototipado evolutivo y el espiral.

A continuación, se explica cada uno de estos modelos.

10.3.1 Desarrollo exploratorio

En el desarrollo exploratorio se presenta al cliente la implementación correspondiente a una parte de los requerimientos, la que se entendió bien. El cliente proporciona sus comentarios y, en base a éstos, se refina el sistema hasta que se logra desarrollar un sistema adecuado a sus necesidades.

10.3.1.1 El desarrollo incremental

En este modelo, el software se entrega en partes pequeñas, pero utilizables, llamadas incrementos. Cada incremento se construye sobre un sistema parcial que ya ha sido entregado. En la Figura 10-2 se ilustra la entrega en incrementos.

Figura 10-2: Desarrollo incremental (adaptado de Pfleeger 2002)

Por ejemplo, supongamos que se construye un sistema de nómina, en el que primero se entrega la parte que captura los datos de los empleados. Luego se agrega la parte que calcula la quincena en base al salario y los impuestos. Posteriormente se le agrega la función de calcular el pago del aguinaldo. Y finalmente el sistema queda completo con la función de calcular el pago de la prima vacacional.

10.3.1.2 El desarrollo iterativo

En el desarrollo iterativo se entrega el esqueleto de un sistema completo desde el principio, y se va agregando funcionalidad con cada nueva versión, como se ilustra en la Figura 10-3.

Figura 10-3: Desarrollo iterativo (adaptado de Pfleeger 2002)

Por ejemplo, se entrega un sistema operativo funcionando completamente (Sistema Operativo para PC). Después se mejora el sistema operativo para que funcione con ventanas (Windows), posteriormente se agrega la capacidad de cambiar el tamaño de las ventanas (Windows 96). Con cada nueva entrega el sistema operativo tiene más facilidades (Windows 98, XP, Vista, 7, 8, 9, 10….). Así la empresa gana dinero con la entrega de una nueva versión.

10.3.1.3 El Prototipado Evolutivo

El Prototipado Evolutivo (ver Figura 10-4) es un modelo de ciclo de vida en el que se desarrolla el concepto del sistema a medida que avanza el proyecto. Normalmente, se comienza desarrollando los aspectos más visibles del sistema. Se presenta la parte ya desarrollada del sistema al cliente y entonces se continúa el desarrollo del prototipo tomando como base la retroalimentación que se recibe del cliente. El ciclo continúa hasta que el prototipo se convierte en el producto final de ingeniería.

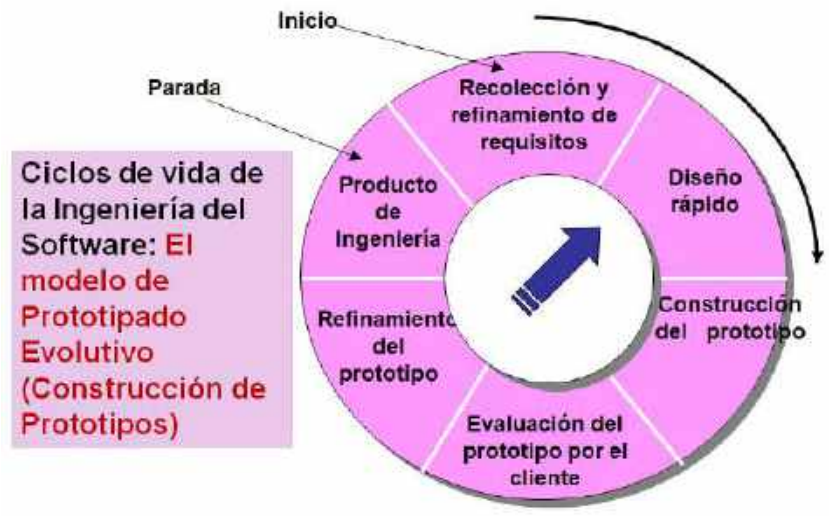


Figura 10-4: El prototipado evolutivo

10.3.1.4 El desarrollo en espiral

El Modelo de Espiral es un modelo de ciclo de vida orientado a los riesgos, dividiendo un proyecto software en miniproyectos. Cada miniproyecto se centra en uno o más riesgos importantes hasta que todos éstos estén controlados. El concepto "riesgo" puede referirse a requerimientos poco comprensibles, arquitecturas poco comprensibles, problemas de ejecución importantes, problemas con la tecnología subyacente, y demás.

El modelo en espiral combina una parte iterativa, que es la construcción de prototipos, con la parte secuencial del modelo en cascada para ir obteniendo resultados parciales en los que se tiene cierto control. Durante las primeras iteraciones, la versión incremental puede ser un modelo en papel o un prototipo. Durante las últimas iteraciones, se producen versiones cada vez más completas del sistema. En la Figura 10-5 se ilustra el desarrollo en espiral enfocado en el proceso de desarrollo de software: análisis, diseño, implementación y pruebas.

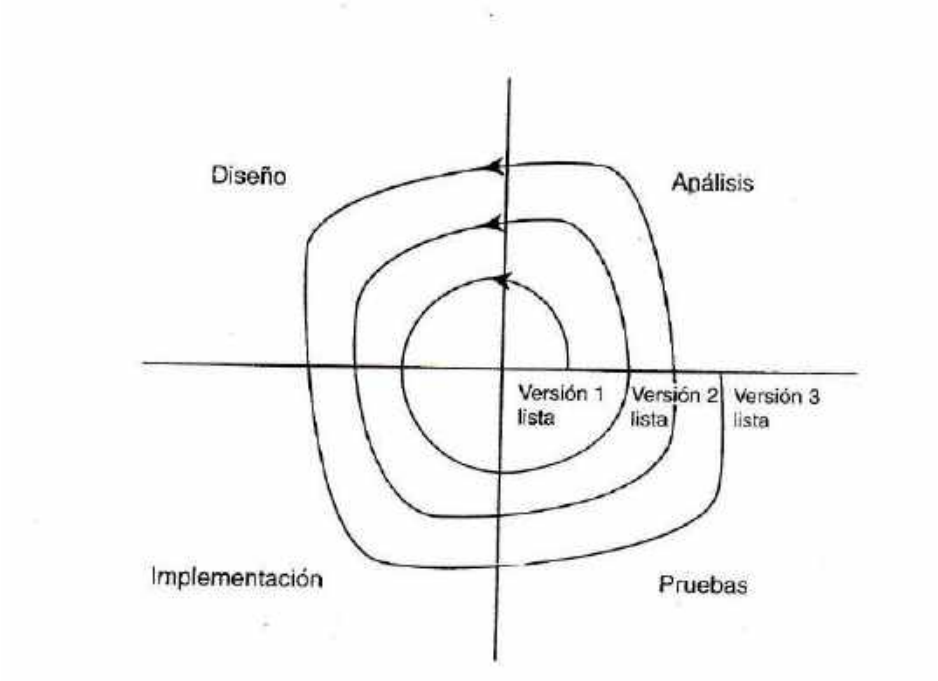


Figura 10-5: Desarrollo en espiral [adaptado de Weitzenfeld, 2004]

En este modelo hay ciclos de trabajo, cada uno de los cuales estudia el riesgo antes de proceder al siguiente ciclo. Cada ciclo comienza con la identificación de los objetivos, soluciones alternativas, restricciones asociadas con cada alternativa y se evalúan para encontrar la mejor. En la Figura 10-6 se muestra el modelo de desarrollo en espiral enfocado en la administración del proyecto.

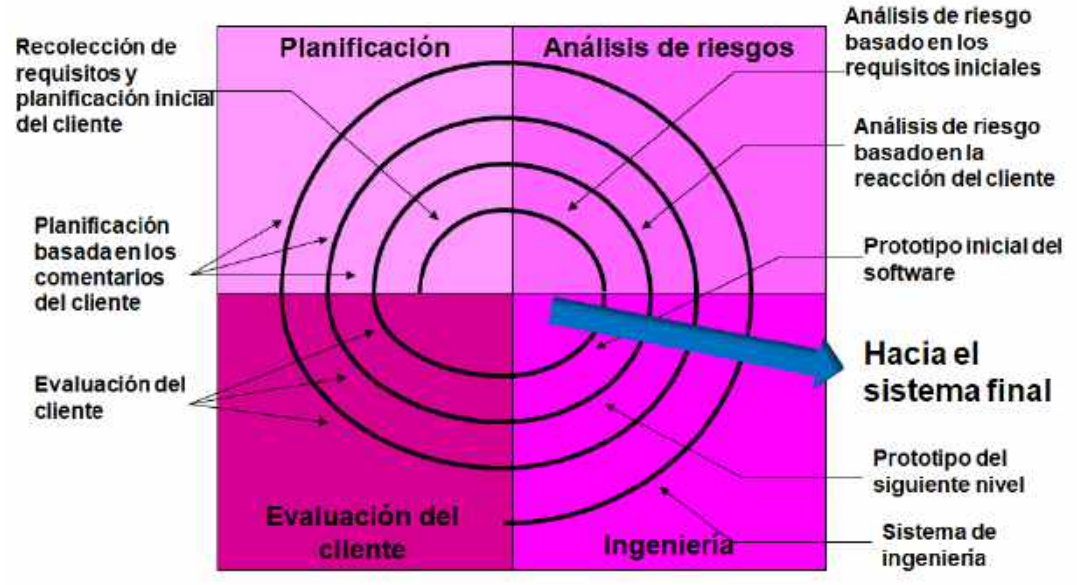


Figura 10-6: El modelo en espiral

10.3.2 Prototipos desechables

El objetivo de los prototipos es comprender los requerimientos del cliente y entonces desarrollar una definición mejorada de los requerimientos para el sistema. El prototipo se centra en experimentar con los requerimientos del cliente que no se comprenden del todo. El software se desarrolla para aprender más sobre un problema o explorar la factibilidad o la conveniencia de las posibles soluciones.

Un prototipo desechable no está pensado para ser utilizado como componente real del software que se entrega al cliente. Un prototipo no es un producto de calidad que deba mantenerse a largo plazo. Por el contrario, los prototipos son creados y probados rápidamente, para luego ser desechados. Sin embargo, es común que, por presiones de tiempo, se trate de enviar un prototipo al mercado como si éste fuera el producto final. En general, siempre existirá un conflicto entre un desarrollo rápido y un producto de calidad.

Es común que los prototipos de la interfaz de usuario no requieran funcionalidad, así que pueden hacerse en papel o con PowerPoint, por ejemplo.

A continuación, se mencionan los puntos clave para que un prototipo tenga éxito o fracase, según [McConnell, 1997].

Claves del éxito:

1.- Se tiene claro el propósito del prototipo y se usa de manera adecuada.

2.- Se comprende la tecnología a utilizarse y su relación con el proceso de prototipos.

3.- Se involucra a tiempo en el proceso a los usuarios finales.

4.- Se está dispuesto a repetir el prototipo para comprender mejor la arquitectura básica.

Puntos que llevan al fracaso:

1.- No se entiende qué es un prototipo y cómo debe usarse.

2.- No se sabe hasta cuándo dejar de evolucionar el prototipo y comenzar de cero. (Puede extenderse demasiado el proceso o terminarse prematuramente).

3.- Se cree que un prototipo razonable es un producto aceptable.

4.- Los prototipos nunca terminan.

En la Figura 10-7 se ilustran las principales características de cada uno de los modelos evolutivos.

Tipo de modelo evolutivo	Subtipo	Características
Desarrollo exploratorio.- Se empieza con las partes del sistema que se comprenden mejor. Se van agregando nuevas características.	Incremental	Entrega el software en partes pequeñas, pero utilizables, llamadas incrementos. Cada incremento se construye sobre un sistema parcial que ya ha sido entregado.
	Iterativo	Se entrega el esqueleto de un sistema completo desde el principio, y se va agregando funcionalidad con cada nueva versión.
	Prototipado Evolutivo	Se construye un prototipo que se entrega al cliente para que dé retroalimentación, y el prototipo se mejora en base a sus comentarios. El ciclo se repite hasta que el cliente acepte el prototipo como la versión final del sistema.
	Espiral	Hay ciclos de trabajo, cada uno de los cuales estudia el riesgo antes de proceder al siguiente ciclo.
Prototipos desechables	El software se desarrolla para aprender más sobre un problema o explorar la factibilidad o la conveniencia de las posibles soluciones.	

Figura 10-7: Características principales de los diferentes modelos evolutivos

10.4 Ventajas y desventajas del desarrollo evolutivo

Los modelos evolutivos permiten mostrar al cliente una versión parcial preliminar que permita obtener retroalimentación y evite problemas con la integración de un código muy grande. Una de las grandes ventajas de los modelos de desarrollo evolutivo es que los clientes pueden comenzar a utilizar un sistema que tiene los requerimientos prioritarios para ponerlo a prueba y reportar sus fallas. De esta manera aumenta la probabilidad de entregar un software que opere satisfactoriamente.

Para poder aplicar un modelo evolutivo se debe poder comenzar con algunos requerimientos prioritarios y dejar para los ciclos posteriores los demás requerimientos, además hay que contar con la participación del usuario, quien debe dedicar tiempo a evaluar y retroalimentar las entregas parciales [Braude, 2007].

Un modelo evolutivo suele ser más efectivo que el modelo en cascada para la producción de sistemas, ya que satisface las necesidades inmediatas de los clientes. Además, tiene la ventaja de que la especificación del sistema se puede desarrollar de forma creciente. Sin embargo, se complica mantener actualizada y correcta la documentación porque cuando los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada versión del sistema. Por otra parte, en sistemas muy grandes, cada nueva adición puede implicar que el código se corrompa debido a una mala administración de los cambios.

Braude (2007) señala un punto importante: "con el propósito de optimizar la productividad en equipo, con frecuencia es necesario comenzar una nueva iteración antes de que la anterior haya terminado", esto no solo dificulta la coordinación de la documentación, sino que dificulta la coordinación de los cambios en los diferentes módulos del sistema cuando un requerimiento tiene impacto en varios de estos módulos.

10.4.1 ¿Cuándo usar modelos evolutivos?

Sommerville (2007) recomienda el uso de modelos evolutivos para sistemas pequeños y de tamaño medio (hasta 500,000 líneas de código), y dice que los problemas del desarrollo evolutivo se hacen particularmente agudos para sistemas grandes y complejos con un período de vida largo, en los cuales diferentes equipos desarrollan distintas partes del sistema.

Cuando se requiere construir un sistema grande, se puede combinar el modelo en cascada con el evolutivo. Así, las partes bien comprendidas se pueden especificar y desarrollar utilizando un proceso basado en el modelo en cascada. Y las otras partes del sistema, como la interfaz de usuario, que son difíciles de especificar por adelantado, desarrollan con el modelo evolutivo-exploratorio.

10.5 Resumen

Los modelos evolutivos tienen como objetivo principal reducir el riesgo y la incertidumbre en el desarrollo. Se utilizan cuando se pueden entregar versiones preliminares con los requerimientos más urgentes o con las partes del sistema que se comprenden mejor. También se utilizan cuando se requiere explorar los requerimientos, porque no se tienen bien definidos. El desarrollo empieza con las partes del sistema que se comprenden mejor y se le van agregando al sistema nuevas características.

Los modelos evolutivos permiten mostrar al cliente una versión parcial preliminar que permita obtener retroalimentación y evite problemas con la integración de un código muy grande. Una de las grandes ventajas de los modelos de desarrollo evolutivo es que los clientes pueden comenzar a utilizar un sistema que tiene los requerimientos prioritarios para ponerlo a prueba y reportar sus fallas. De esta manera aumenta la probabilidad de entregar un software que opere satisfactoriamente.

Para poder aplicar un modelo evolutivo se debe poder comenzar con algunos requerimientos prioritarios y dejar para los ciclos posteriores los demás requerimientos, además hay que contar con la participación del usuario, quien debe dedicar tiempo a evaluar y retroalimentar las entregas parciales.

Un modelo evolutivo suele ser más efectivo que el modelo en cascada para la producción de sistemas, ya que satisface las necesidades inmediatas de los clientes. Además, tiene la ventaja de que la especificación del sistema se puede desarrollar de forma creciente. Sin embargo, se complica mantener actualizada y correcta la documentación porque cuando los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada versión del sistema. Por otra parte, en sistemas muy grandes, cada nueva adición puede implicar que el código se corrompa debido a una mala administración de los cambios.

10.6 Cuestionario

1.- ¿Cuáles son las ventajas de ir entregando versiones parciales al cliente?

2.- ¿Cuándo se puede utilizar modelos evolutivos?

3.- ¿Cómo se clasifican los modelos evolutivos según Sommerville?

4.- ¿Cuáles son las **ventajas** del desarrollo evolutivo?

5.- ¿Cuáles son las **desventajas** del desarrollo evolutivo?

11 Segundo Caso de Estudio: Desarrollo Evolutivo del Sistema II

11.1 Objetivos del capítulo

Objetivo general:

- Aplicar los conceptos estudiados en el capítulo anterior, en el desarrollo de un sistema evolutivo sencillo.

Objetivos específicos:

- Comprender, mediante un ejemplo práctico, la forma en la que se atienden los requerimientos en el desarrollo evolutivo.
- Conocer, con un ejemplo, el ciclo de diseño-codificación-pruebas para cada conjunto de requerimientos del desarrollo evolutivo.
- Comprender la conveniencia de diseñar antes de codificar.

11.2 El Sistema II: Control escolar de la Academia Patito.

En este capítulo desarrollaremos el **Sistema II** el cual servirá para el *Control escolar de la Academia Patito*. Este sistema no requiere interfaz gráfica, es decir, funciona en la consola. A continuación, se presentan los requerimientos del sistema completo y, en las siguientes secciones se estudiará el desarrollo del sistema por etapas.

11.2.1 Visión global de los Requerimientos del Sistema II

Con el ***Sistema II*** se pueden hacer las siguientes operaciones:

- Alta, consulta y baja de alumnos
- Grupos (Alta y consulta de grupos)
- Inscripciones (alta y baja de alumno en grupo)
- Calificaciones (Consultar matrículas y calificación, asentar calificación)

El sistema terminará de ejecutarse cuando el usuario seleccione la opción "salir" del menú principal.

Figura 11-1: Diagrama de casos de uso del Sistema II

En la Figura 11-1 se ilustra el diagrama de casos de uso del Sistema II. Basándonos en este diagrama describimos los menús del sistema. Primero se presentará un *menú principal* con las siguientes opciones:

Selecciona la opción deseada:

1.- Alumnos

2.- Grupos

3.- Inscripciones

4.- Calificaciones

5.- Salir

Habrá un *menú de alumnos* que desplegará lo siguiente:

Selecciona la opción deseada:

1.- Dar de alta a un alumno

2.- Consultar alumnos

3.- Dar de baja a un alumno

4.- Regresar

También habrá un *menú de grupos* que desplegará lo siguiente:

Selecciona la opción deseada:

1.- Dar de alta un nuevo grupo

2.- Consultar los grupos

3.- Regresar

Habrá también un *menú de inscripciones* que desplegará lo siguiente:

Selecciona la opción deseada:

1.- Dar de alta a un alumno en un grupo

2.- Dar de baja a un alumno en un grupo

3.- Regresar

Por último, habrá un *menú de calificaciones* que desplegará lo siguiente:

Selecciona la opción deseada:

1.- Asentar calificación a un alumno

2.- Consultar matrículas y calificaciones de un grupo

3.- Regresar

En la Figura 11-2 se ilustra el Diagrama de Transición entre Interfaces de Usuario (DTIU) del Sistema II.

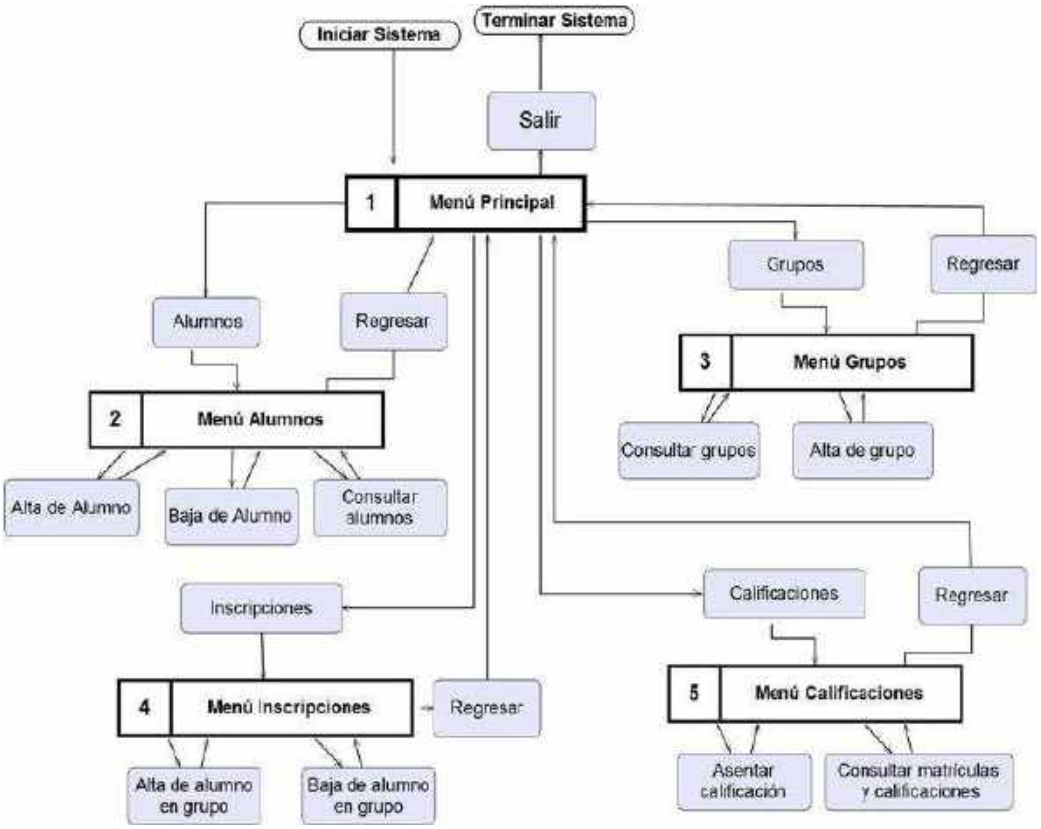


Figura 11-2: Diagrama de Transición entre Interfaces de Usuario (DTIU) del Sistema II

Desarrollaremos el **Sistema II** con un modelo incremental en 4 etapas. Comenzaremos con la etapa 1 como se ilustra en la Figura 11-3.

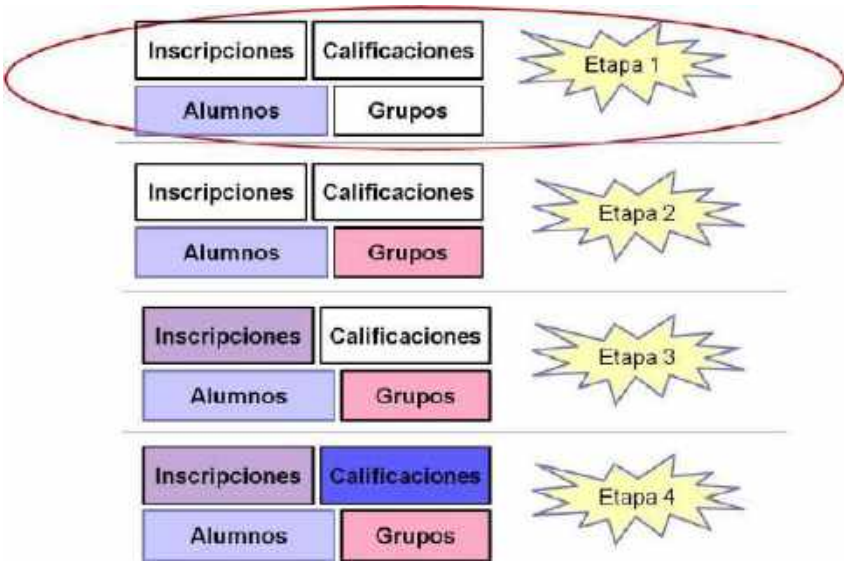


Figura 11-3: Las 4 etapas del desarrollo incremental del Sistema II

En la siguiente sección comenzaremos con la primera etapa.

11.3 Etapa 1 del desarrollo iterativo

11.3.1 Requerimientos a desarrollar en la primera etapa: *funcionamiento del menú alumnos*

R.1.- Funcionamiento de la opción "Alta de alumno".

R1.1.- Al seleccionar "Alta de alumno" se solicita al usuario el nombre completo del alumno, y su teléfono. Posteriormente se genera una matrícula y se guardan todos sus datos en un archivo llamado "Alumnos.txt".

R1.2.- Se notifica al usuario la matrícula asignada cuando la operación haya sido exitosa.

R.2.- Funcionamiento de la opción "Baja de alumno".

R2.1.- Al seleccionar "Baja de alumno" se solicita su matrícula, se busca la matrícula en el archivo y se borra al alumno del archivo.

R2.2.- En caso de que no se encuentre la matrícula del alumno que se desea dar de baja, se indicará al usuario.

R2.3.- Se notificará al usuario cuando la operación de alta o baja haya sido exitosa.

R.3.- Funcionamiento de la opción "Consultar alumnos".

R3.1.- Al seleccionar "Consultar Alumnos" se desplegarán:

- la matrícula
- el nombre
- el teléfono

de todos los alumnos existentes.

11.3.2 Diseño para la primera etapa

En esta etapa elaboraremos el esqueleto del sistema. Utilizaremos la clase abstracta Menu ( ) diseñada para la reutilización de código. Así, los menús del sistema son descendientes de esta clase, como se ilustra en la Figura 11-4.

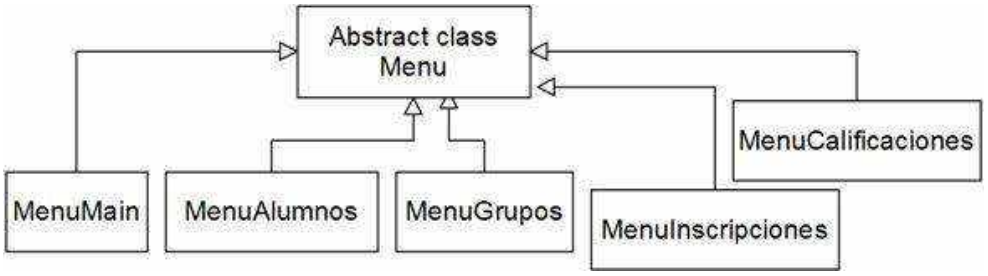


Figura 11-4: Diagrama de clases de los menús del sistema

Además del esqueleto del sistema, en esta etapa desarrollaremos la funcionalidad del "Menú Alumnos" (ver Figura 11-5).

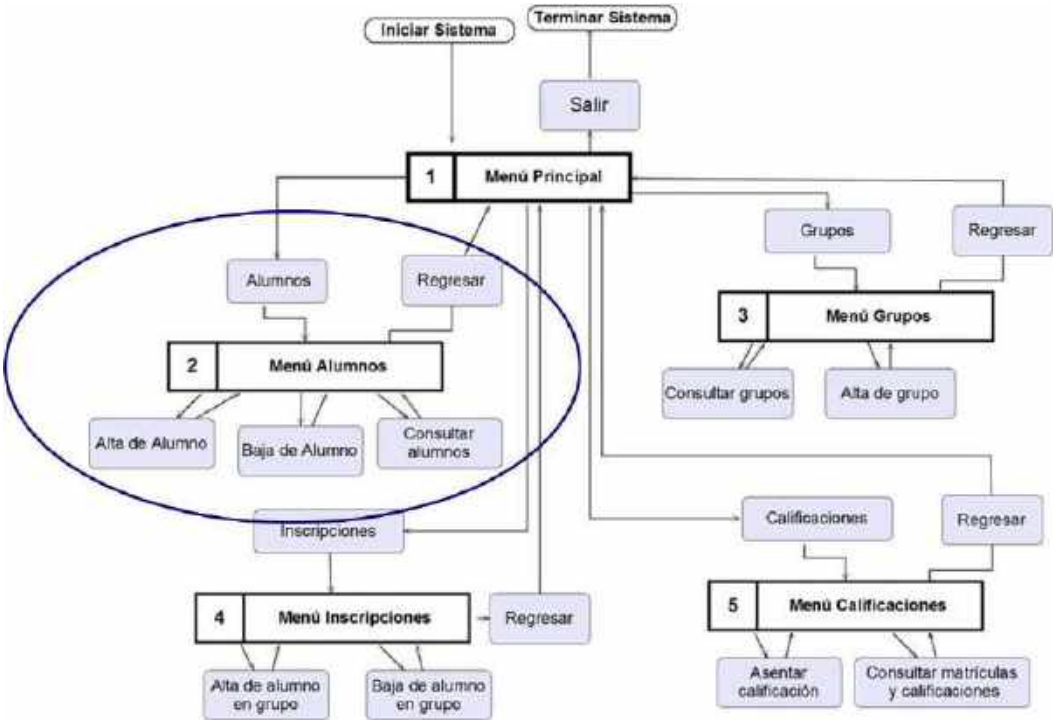


Figura 11-5: En la primera etapa se desarrolla funcionalidad para el Menú Alumnos

La clase que se necesita para esta funcionalidad es `Alumno`. En la Figura 11-6 se describen sus atributos y sus métodos.

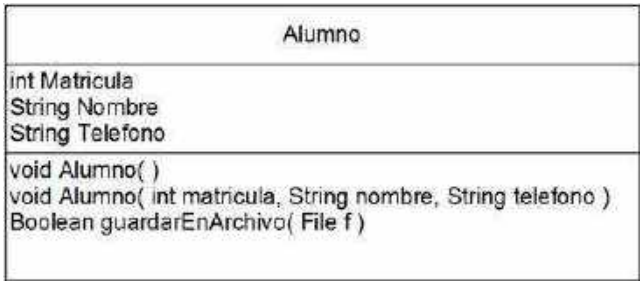


Figura 11-6: La clase `Alumno`

El algoritmo de funcionamiento para *dar de alta a un alumno* se ilustra en el diagrama de flujo de la Figura 11-7.

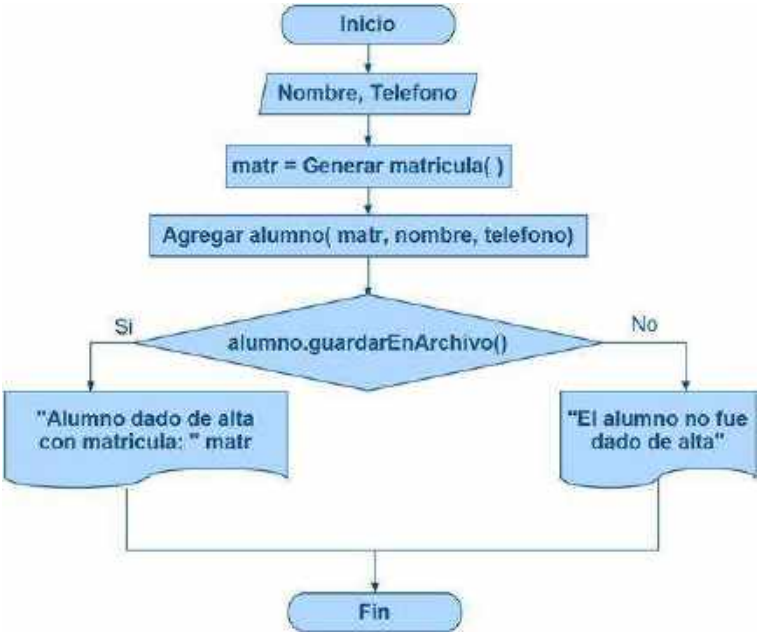


Figura 11-7: Diagrama de flujo del algoritmo para dar de alta a un alumno

Como se aprecia en el diagrama de flujo anterior, se requiere generar una matrícula cada vez que se da de alta a un alumno. En la Figura 11-8 se ilustra el diagrama de flujo con el algoritmo para generar una nueva matrícula.

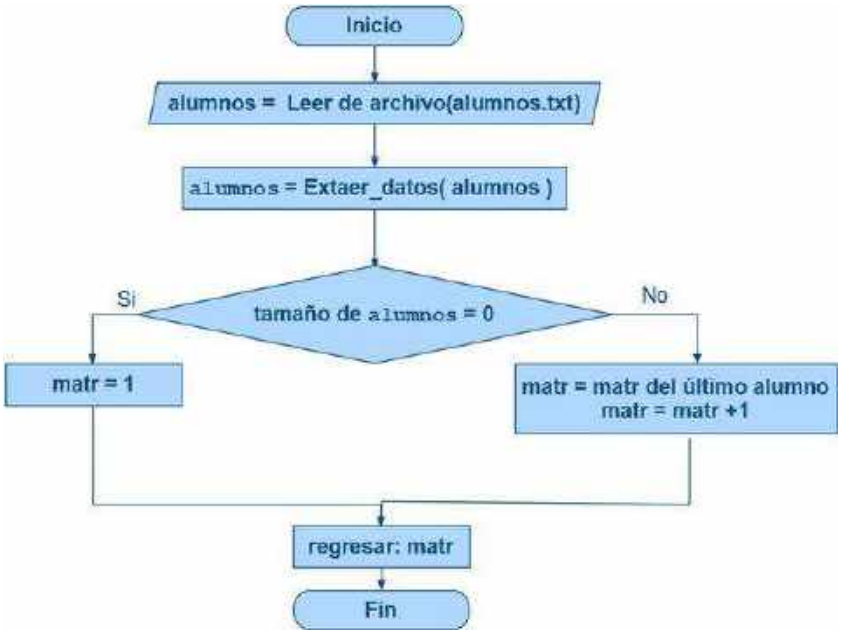


Figura 11-8: Diagrama de flujo del algoritmo para generar una matrícula

El algoritmo de funcionamiento para ***dar de baja a un alumno*** se ilustra en el diagrama de flujo de la Figura 11-9.

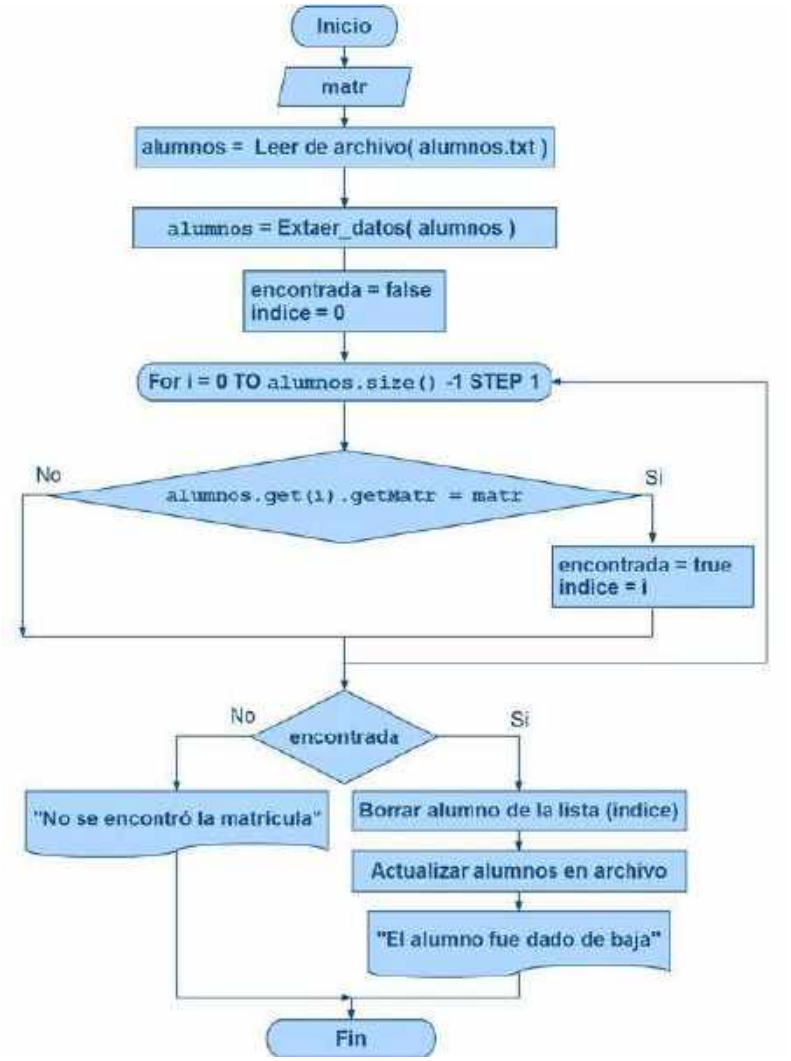


Figura 11-9: Diagrama de flujo del algoritmo para dar de baja a un alumno

11.3.3 Codificación y pruebas de validación de la primera etapa

11.3.3.1 Codificación del primer bloque (el esqueleto)

La siguiente clase abstracta llamada `Menu( )` se reutiliza con las clases hijas:

- `MenuMain( )`
- `MenuAlumnos( )`
- `MenuGrupos( )`
- `MenuIscripciones ( )`
- `MenuCalificaciones ( )`

```
package utilerias;
import java.util.Scanner;
public abstract class Menu {
```

```
  // Las clases descendientes de Menu deberán inicializar las opciones
  // opcion[0] es la opción 1 del menú en pantalla.
  public String [] opciones;
  public int opcion(){
    if( opciones.length < 1 ) return 0;
    System.out.println();
    for( int i=0; i<opciones.length; i++ )
      System.out.println( i+1 + ":" + opciones[i] );
    System.out.println();
    int opcion;
    Scanner input = new Scanner( System.in );
    do{
      System.out.println("Escribe la opción deseada:");
      opcion = input.nextInt();  // lee la opción que elige el usuario
    }while( opcion<1 || opcion>opciones.length );
    return opcion;
  }
}
```

A continuación presentamos las clases hijas de `Menu()`

```
package sistemaescuela;
import utilerias.Menu;
public class MenuMain extends Menu{
    MenuMain(){
          opciones = new String[5];
          opciones[0] = "Alumnos";
          opciones[1] = "Grupos";
          opciones[2] = "Inscripciones";
          opciones[3] = "Calificaciones";
          opciones[4] = "Salir";
       }
}
```

```
package sistemaescuela;
import utilerias.Menu;
public class MenuAlumnos extends Menu{
    MenuAlumnos(){
          opciones = new String[4];
          opciones[0] = "Dar de alta a un alumno";
          opciones[1] = "Consultar alumnos";
          opciones[2] = "Dar de baja a un alumno";
          opciones[3] = "Regresar";
       }
}
```

```
package sistemaescuela;
import utilerias.Menu;
public class MenuGrupos extends Menu{
    MenuGrupos(){
        opciones = new String[3];
         opciones[0] = "Dar de alta a un nuevo grupo";
         opciones[1] = "Consultar los grupos";
         opciones[2] = "Regresar";
    }
}
```

```
package sistemaescuela;
import utilerias.Menu;
```

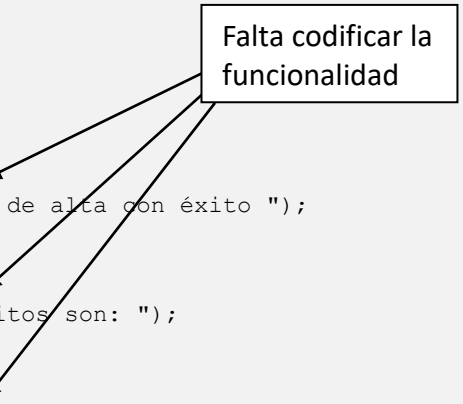
```
public class MenuInscripciones extends Menu{
    MenuGrupos(){
        opciones = new String[3];
         opciones[0] = "Dar de alta a un alumno en un grupo";
         opciones[1] = "Dar de baja a un alumno en un grupo";
         opciones[2] = "Regresar";
    }
}
```

```
package sistemaescuela;
import utilerias.Menu;
public class MenuCalificaciones extends Menu {
    MenuCalificaciones(){
         opciones = new String[3];
         opciones[0] = "Asentar calificación a un alumno";
         opciones[1] = "Consultar matrículas y calificaciones de un
                        grupo";
         opciones[2] = "Regresar";
       }
}
```

Primero codificaremos el *esqueleto de los menús* `main` y `alumnos`, sin funcionalidad.

```
package sistemaescuela;
public class SistemaEscuela {
    /**
     * Sistema que administra la escuela "Patito"
     */
    public static void main(String[] args) {
        MenuMain  menuMain = new MenuMain();
        while( true ){
            int opcion = menuMain.opcion();

            switch( opcion ){
                case 1: alumnos();
                        break;
                case 2: grupos();
                        break;
                     case 3: inscripciones();
                        break;
                case 4: calificaciones();
                        break;
                case 5: return;
            }
       }
    }
```

```
private static void alumnos() {
    MenuAlumnos  menuAlumnos = new MenuAlumnos();
    while( true ){
        int opcion = menuAlumnos.opcion();
        switch( opcion ){
            case 1: altaAlumno();
                    break;
            case 2: consultarAlumnos();
                    break;
            case 3: bajaAlumno();
                    break;
            case 4: return;
        }
    }
}

private static void altaAlumno() {
    System.out.println("El alumno fue dado de alta con éxito ");
}

private static void consultarAlumnos() {
    System.out.println(" Los alumnos inscritos son: ");
}

private static void bajaAlumno() {
    System.out.println("El alumno fue dado de baja");
}
```



Ahora verificamos el funcionamiento del flujo de los menús. En la Figura 11-10 se observa que el primer bloque compila y funciona correctamente.

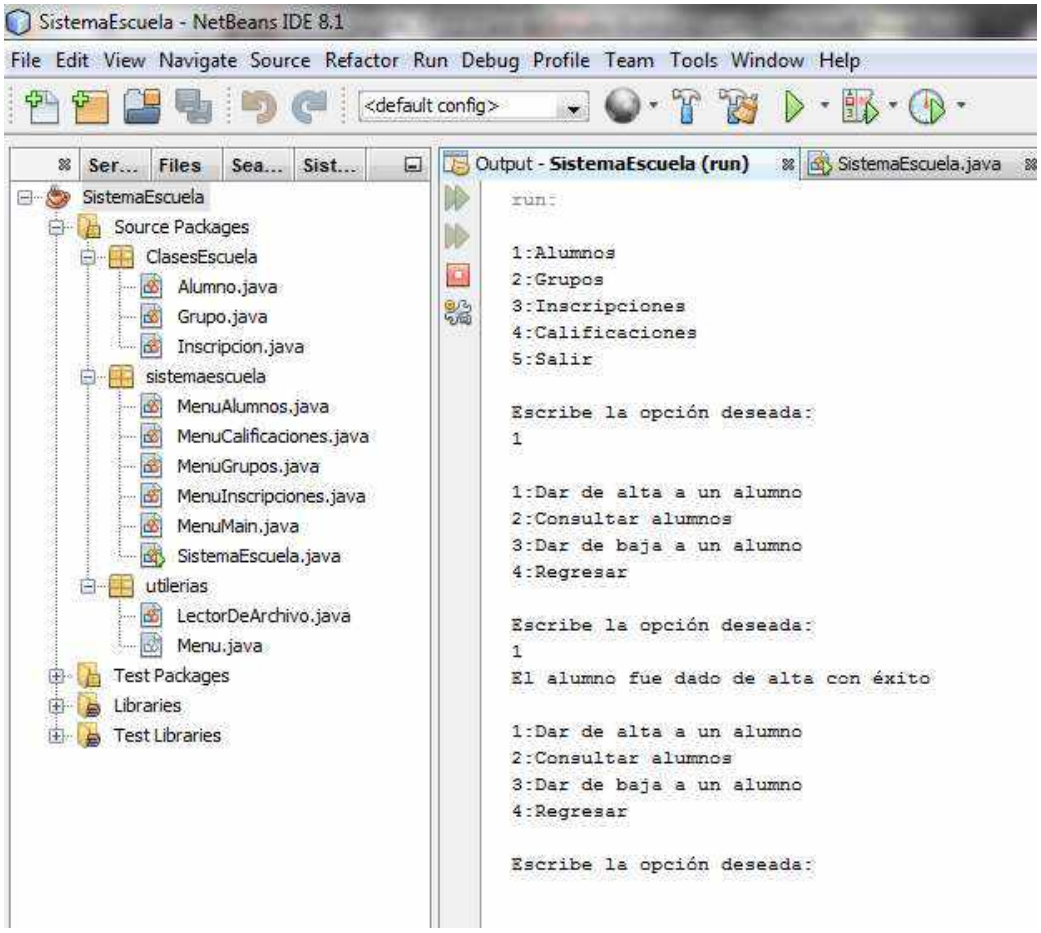


Figura 11-10: Funcionamiento correcto del primer bloque de código

11.3.3.2 Desarrollo del segundo bloque de la etapa 1

En este segundo bloque codificaremos el esqueleto de los menús para Grupos, Inscripciones y Calificaciones, sin funcionalidad.

```
private static void grupos() {
    MenuGrupos menuGrupos = new MenuGrupos();
    while( true ){
       int opcion = menuGrupos.opcion();

       switch( opcion ){
           case 1: altaGrupo();
                   break;
           case 2: consultaGrupos();
                   break;
           case 3: return;
       }
    }
}
```

```
private static void altaGrupo() {
    System.out.println("El grupo se creó con éxito");
}

private static void consultaGrupos() {
    System.out.println("Los grupos existentes son:");
}
```

```
private static void inscripciones(  ) {
    MenuInscripciones menuInscripciones = new MenuInscripciones();
    while( true ){
       int opcion = menuInscripciones.opcion();

       switch( opcion ){
           case 1: altaAlumnoEnGrupo();
                   break;
           case 2: bajaAlumnoEnGrupo();
                   break;
           case 3: return;
       }
    }
}

private static void altaAlumnoEnGrupo() {
    System.out.println("El alumno quedó inscrito en el grupo");
}

private static void bajaAlumnoEnGrupo() {
    System.out.println("El alumno fue dado de baja del grupo");
}
```

```
private static void calificaciones() {
    MenuCalificaciones menuCalificaciones = new MenuCalificaciones();

    while( true ){
        int opcion = menuCalificaciones.opcion();

        switch( opcion ){
            case 1: calificarAlumno();
                    break;
            case 2: Matriculas_Calific_DeGrupo();
                    break;
            case 3: return;
        }
    }
}

private static void calificarAlumno() {
    System.out.println("La calificación quedó asentada");
}

private static void Matriculas_Calific_DeGrupo () {
    System.out.println("Las matrículas y calificaciones son:");
}
```

En la Figura 11-11 se ilustran pruebas de los menús grupos e inscripciones.

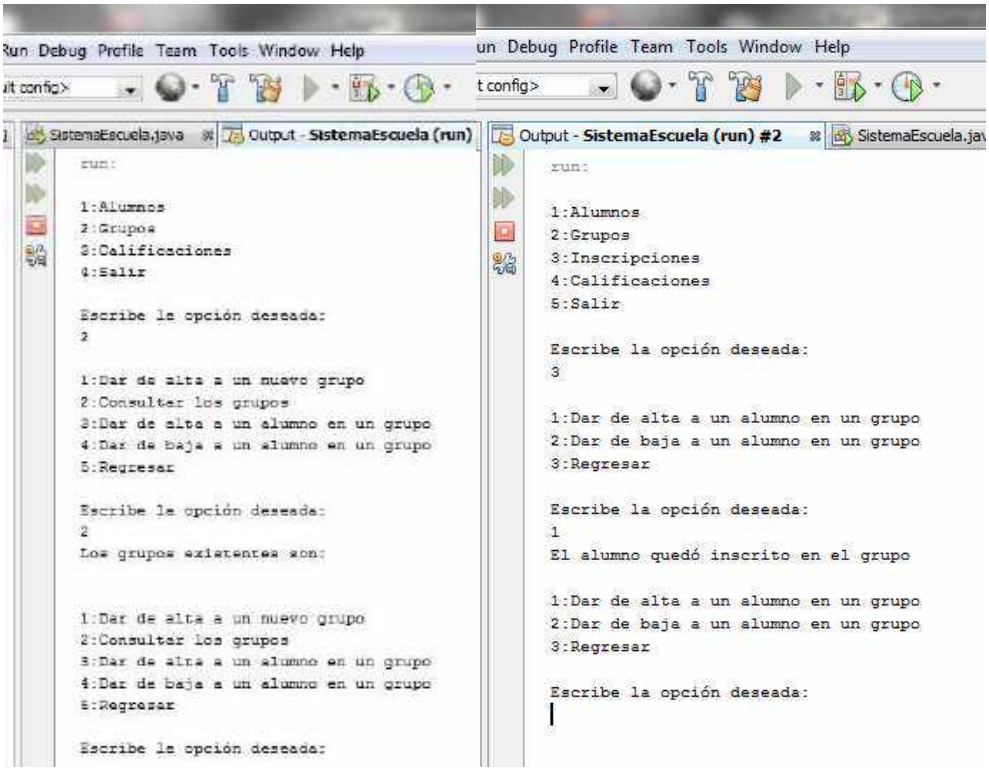


Figura 11-11: Funcionamiento correcto del segundo bloque de código (opciones 2 y 3)

En la Figura 11-12 ilustran pruebas del menú calificaciones. Se observa que el segundo bloque compila y funciona correctamente.

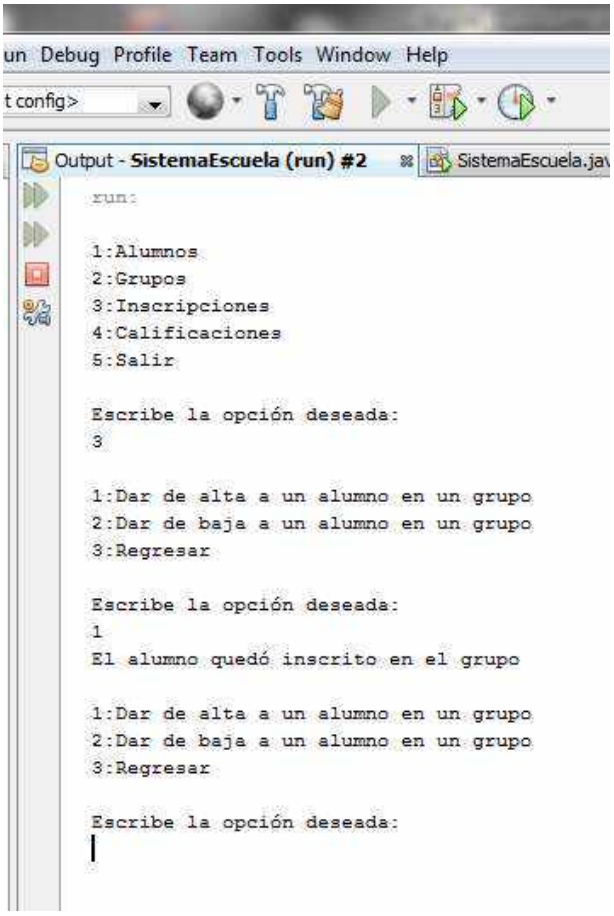


Figura 11-12: Funcionamiento correcto del segundo bloque de código opción 4

11.3.3.3 Desarrollo del tercer bloque de la etapa 1

En el tercer bloque agregaremos la funcionalidad ***Alta de alumno*** del menú "Alumnos". Primero codificamos la clase Alumno.

```
package ClasesEscuela;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class Alumno {
    int    matricula;
    String nombre;
    String telefono;

    public Alumno(){
    }

    public Alumno(int matricula, String nombre, String telefono) {
        this.matricula = matricula;
        this.nombre = nombre;
```

```
        this.telefono = telefono;
    }

    public Boolean guardarEnArchivo( File f ){
       Boolean exito = false;
       try{
         FileWriter w = new FileWriter( f, true );

         BufferedWriter bw = new BufferedWriter(w);
         PrintWriter salida = new PrintWriter( bw );
         //Escribimos los datos en archivo separando los campos por comas
         salida.println( matricula + ", " + nombre + ", " + telefono );
         salida.close();
         bw.close();
         exito = true;
       }catch(IOException e){};
         return exito;
       }
    ... getters...
    ... Setters...
}
```

append = true para agregar sin sobre-escribir

Instanciamos un objeto de clase `File` con el nombre de archivo "Alumnos.txt" en la clase main.

```
public class SistemaEscuela {
    /**
     * Sistema que administra la escuela "Patito"
     */
    public static void main(String[] args) {
        MenuMain  menuMain = new MenuMain();

        File f = new File( "Alumnos.txt" );

        while( true ){
            int opcion = menuMain.opcion();

            switch( opcion ){
                case 1: alumnos( f );
                        break;
                case 2: grupos();
                        break;
                case 3: inscripciones();
                        break;
                case 4: calificaciones();
                        break;
                case 5: return;
            }
        }
    }
```

Agregamos el parámetro de clase `File` en el método del menú alumnos y lo enviamos como parámetro al método `altaAlumno()`

```
public class SistemaEscuela {
    /**
     * Sistema que administra la escuela "Patito"
     */
    . . .
    private static void alumnos( File f ) {
        MenuAlumnos  menuAlumnos = new MenuAlumnos();
```

```
        while( true ){
            int opcion = menuAlumnos.opcion();

            switch( opcion ){
                case 1: altaAlumno( f );
                        break;
                case 2: consultarAlumnos();
                        break;
                case 3: bajaAlumno();
                        break;
                case 4: return;
            }
        }
    }
```

Ahora nos basamos en el diagrama de flujo de la Figura 11-7 para codificar el método `altaAlumno()`

```
public class SistemaEscuela {
    . . .
    private static void altaAlumno( File f ) {
        int    matricula;
        String nombre;
        String telefono;
        Scanner input = new Scanner( System.in );

        System.out.println( "Nombre del alumno? ");
        nombre = input.nextLine();
        System.out.println( "Telefono? ");
        telefono = input.nextLine();

        matricula = generaMatricula();

        Alumno alumno = new Alumno( matricula, nombre, telefono );

         if( alumno.guardarEnArchivo( f ) )
               System.out.println("El alumno fue dado de alta con la
                                   matrícula:" + matricula);
         else
               System.out.println("El alumno no fue dado de alta ");

   }
```

Y para codificar el método `generarMatricula()` usamos como guía el diagrama de flujo de la Figura 11-8.

```
private static int generaMatricula() {
    int matr = 0;
    ArrayList<String> alumnosString = new ArrayList<String>();
    ArrayList<Alumno> alumnos = new ArrayList<Alumno>();

    LectorDeArchivo lectorDeArchivo = new LectorDeArchivo("Alumnos.txt");
    alumnosString = lectorDeArchivo.LeerArchivo();

    // extraer las cadenas de caracteres a objetos
    alumnos = extraerDatosAlumno( alumnosString );

    if( alumnos.size() == 0 )
        matr = 1;
```

```
    else {
        matr = alumnos.get(alumnos.size()-1).getMatricula();
        matr = matr +1:
    }
    return (matr);
}
```

A continuación presentamos el código de `extraerDatosAlumno( )`

```
private static ArrayList<Alumno> extraerDatosAlumno( ArrayList<String>
                                                        alumnosString) {
    ArrayList<Alumno> alumnos = new ArrayList<Alumno>();
    Alumno alumno;
    String linea=null;
    String [] tokensLinea = null;
    String matr;
    tokensLinea = new String[8];

    for( int i=0; i < alumnosString.size() ; i++ ){
      // Se lee una línea completa, luego se separan los tokens
      // y se guardan en un arreglo de Strings.
      linea= alumnosString.get(i);

      alumno = new Alumno();
      //Cuando se escribió en el archivo los campos se separaron con coma
      tokensLinea = linea.split(",");

      matr = tokensLinea[0];
      alumno.setMatricula( Integer.parseInt(matr) );
      alumno.setNombre( tokensLinea[1] );
      alumno.setTelefono( tokensLinea[2] );

      alumnos.add(alumno);
    }
    return alumnos;
}
```

Necesitaremos la clase `LectorDeArchivo`, esta clase la agregaremos en el paquete utilerías.

```
package utilerias;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.ArrayList;
public class LectorDeArchivo {
    private String nombreArchivo;

    public LectorDeArchivo( String nombreArchivo ){
       this.nombreArchivo = nombreArchivo;
    }
    public ArrayList<String> LeerArchivo(  ){
       ArrayList<String> lectura = new ArrayList<String>();
       File archivo = null;

       FileReader fr = null;
       BufferedReader br = null;
       try {// Apertura del archivo y creacion de BufferedReader para
            // poder leer con el metodo readLine()).
          archivo = new File ( nombreArchivo );
          fr = new FileReader (archivo);
          br = new BufferedReader(fr);
```

```
          // Lectura del archivo
          String linea;
          while((linea=br.readLine())!=null)
            lectura.add(linea);
        }
        catch(Exception e){
           e.printStackTrace();
        }finally{
          // se cierra el archivo tanto para el caso normal
          //  como para la excepción
          try{
            if( null != fr ){
               fr.close();
            }
          }catch (Exception e2){
            e2.printStackTrace();
          }
        }
       return lectura;
    }
}
```

11.3.3.4 *Pruebas de validación del tercer bloque de código.*

I.- Pruebas del menú alumnos opción: **Alta de alumno**

I.1.- Se selecciona la opción 1 del menú principal. Luego la opción 1 del menú de alumnos. Se darán los siguientes datos:

```
Nombre: Mario García
Teléfono: 58692221
```

El sistema deberá proporcionar una matrícula asignada.

I.2.- Se selecciona la opción 2 del menú de alumnos, deberá aparecer el alumno Mario García con su teléfono y su matrícula = 1.

I.3.- Repetir las pruebas I.1 y I.2 con cada uno de los siguientes datos:

```
Elena Perez, 5896-2212
Jose Aguilar, 3975-9075
Juan Sanchez, 9807-6518
Silvia Lopez, 98123-0012
```

El sistema debe asignar una matrícula en orden sucesivo, el archivo "alumnos.txt" debe contener:

```
1, Mario Garcia, 58692221
2, Elena Perez, 5896-2212
3, Jose Aguilar, 3975-9075
4, Juan Sanchez, 9807-6518
5, Silvia Lopez, 98123-0012
```

Después de ejecutar las pruebas de validación verificamos que se generó correctamente el archivo "alumnos.txt" que está en la carpeta del proyecto, como se ilustra en la Figura 11-13.

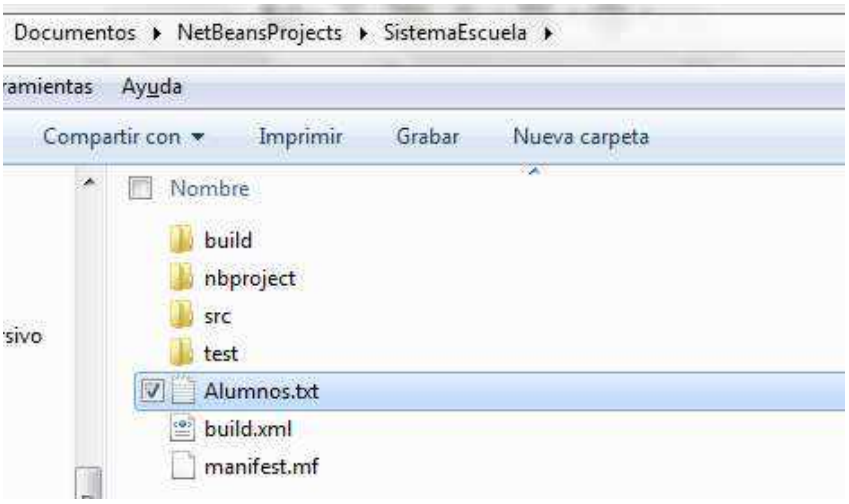


Figura 11-13: Archivo "Alumnos.txt" en la carpeta del proyecto

Cuando abrimos el archivo "alumnos.txt" desde NetBeans podemos verificar su contenido, como se ilustra en la Figura 11-14.

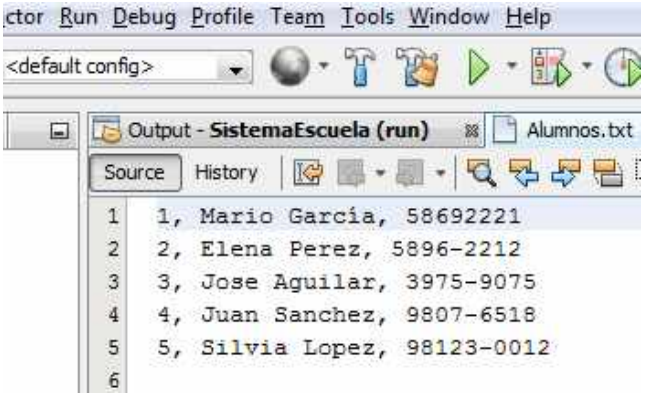


Figura 11-14 : Contenido del archivo "alumnos.txt"

11.3.3.5 Desarrollo del cuarto bloque de la etapa 1

En el cuarto bloque agregaremos la funcionalidad ***Consultar alumnos*** del menú "Alumnos". El código del método `consultarAlumnos( )` es el siguiente:

```
private static void consultarAlumnos() {
    ArrayList<String> alumnos = new ArrayList<String>();
    LectorDeArchivo lectorDeArchivo = new LectorDeArchivo("Alumnos.txt");
    alumnos = lectorDeArchivo.LeerArchivo();

    System.out.println(" Los alumnos inscritos son: ");
    for( int i=0; i< alumnos.size() ; i++){
         System.out.println(alumnos.get(i));
    }
}
```

11.3.3.6 Pruebas de validación del cuarto bloque de código

II.- Pruebas de validación del *menú alumnos* opción: **Consultar alumnos**

II.1.- Se selecciona la opción 1 del menú principal. Luego la opción 2 del menú de alumnos. El sistema deberá desplegar correctamente los datos que están en el archivo "Alumnos.txt".

En la Figura 11-15 se ilustra que el cuarto bloque compila y funciona correctamente.

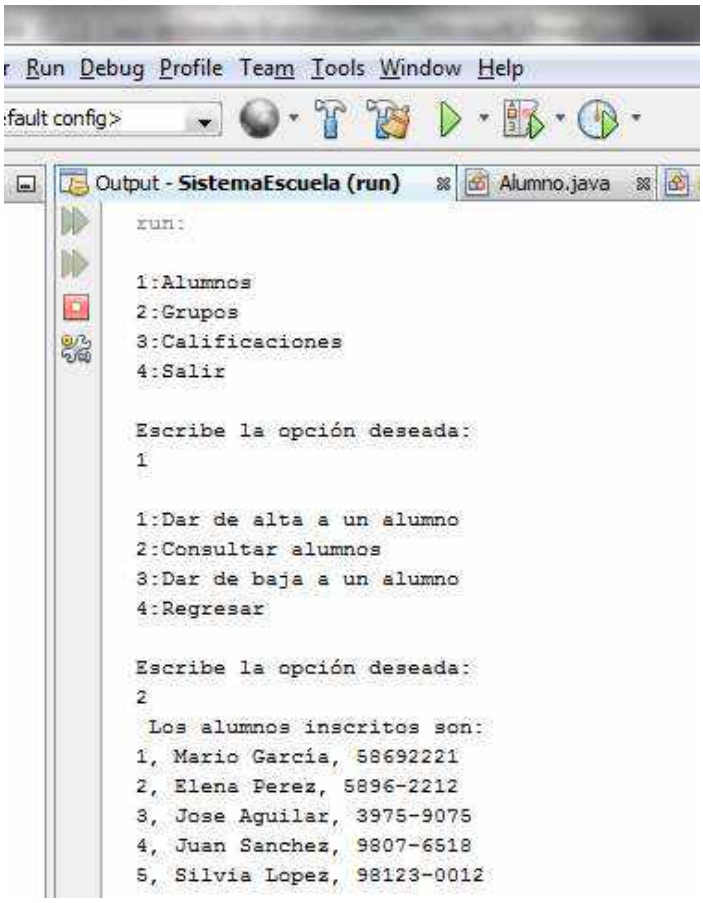


Figura 11-15: Funcionamiento correcto del cuarto bloque de código

11.3.3.7 Desarrollo del quinto bloque de la etapa 1

En el quinto bloque agregaremos la funcionalidad ***Baja de alumno*** del menú "Alumnos". Nos basamos en el diagrama de flujo de la Figura 11-9 para codificar el método `bajaAlumno()`

```
private static void bajaAlumno() {
    int matricula;
    Scanner input = new Scanner( System.in );
    ArrayList<String> alumnosString = new ArrayList<String>();
    ArrayList<Alumno> alumnos = new ArrayList<Alumno>();
```

```
    LectorDeArchivo lectorDeArchivo = new LectorDeArchivo("Alumnos.txt");
    alumnosString = lectorDeArchivo.LeerArchivo();

    alumnos = extraerDatos( alumnosString );
    System.out.println( "Matricula del alumno a dar de baja? ");
    matricula = input.nextInt();

    Boolean encontrada = false;
    int indice = 0;
    for( int i=0; i < alumnos.size() ; i++ ){
         if( alumnos.get(i).getMatricula() == matricula ){
             encontrada = true;
             indice = i;
         }
    }
    if( encontrada ){
        alumnosString.remove(indice);
        alumnos.remove( indice );
        actualizarAlumnos( alumnosString, alumnos );
        System.out.println("El alumno fue dado de baja");
    }
    else
        System.out.println("No se encontró la matrícula proporcionada");
    }
}
```

Codificamos el método `extraerDatos( )`:

```
private static ArrayList<Alumno> extraerDatos( ArrayList<String>
                                                alumnosString) {
    ArrayList<Alumno> alumnos = new ArrayList<Alumno>();
    Alumno alumno;
    String linea=null;
    String [] tokensLinea = null;
    String matr;
    tokensLinea = new String[8];

    for( int i=0; i < alumnosString.size() ; i++ ){
       // Se lee una línea completa, luego se separan los tokens
       // y se guardan en un arreglo de Strings.
       linea= alumnosString.get(i);
       alumno = new Alumno();
       // Cuando se escribió en el archivo los campos se separaron
       // con coma
       tokensLinea = linea.split(",");
       matr = tokensLinea[0];
       alumno.setMatricula( Integer.parseInt(matr) );
       alumno.setNombre( tokensLinea[1] );
       alumno.setTelefono( tokensLinea[2] );

       alumnos.add(alumno);
   }
   return alumnos;
}
```

Codificamos el método `actualizarAlumnos( )`:

```
private static void actualizarAlumnos( ArrayList<String> alumnosString,
                                       ArrayList<Alumno> alumnos ) {
    File file = new File( "Alumnos.txt" );
    File archivo = file;
    file.delete();  // Se borra el archivo anterior
```

```
    Alumno alumno = new Alumno();

    for( int i=0; i < alumnos.size(); i++ ){
         alumno = alumnos.get(i);
         // Se guardan en "Alumnos.txt" los datos actualizados
         alumno.guardarEnArchivo( archivo, alumnosString.get(i));
    }

}
```

11.3.3.8 *Pruebas de validación del quinto bloque de código.*

I.- Pruebas del menú alumnos opción: **Baja de alumno**

I.1.- Se selecciona la opción 1 del menú principal. Luego la opción 3 del menú de alumnos. Proporcionar el número de matrícula = 4. Posteriormente volver a consultar a los alumnos.

Ya no debe aparecer el alumno con matrícula 4 en la lista, como se observa en la Figura 11-16.

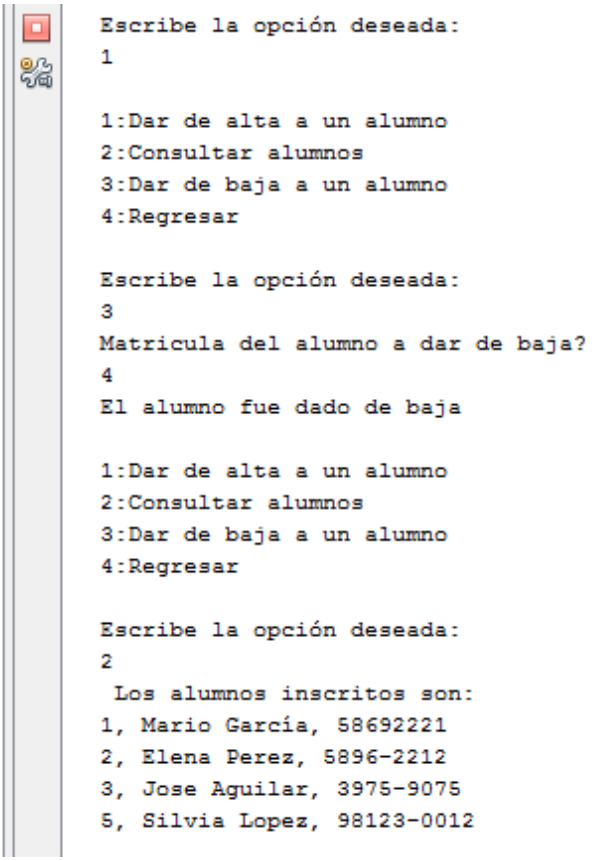


Figura 11-16: Funcionamiento correcto del quinto bloque de código

11.4 Etapa 2 del desarrollo incremental

En esta segunda etapa desarrollaremos la funcionalidad del menú "Grupos", como se ilustra en la Figura 11-17.

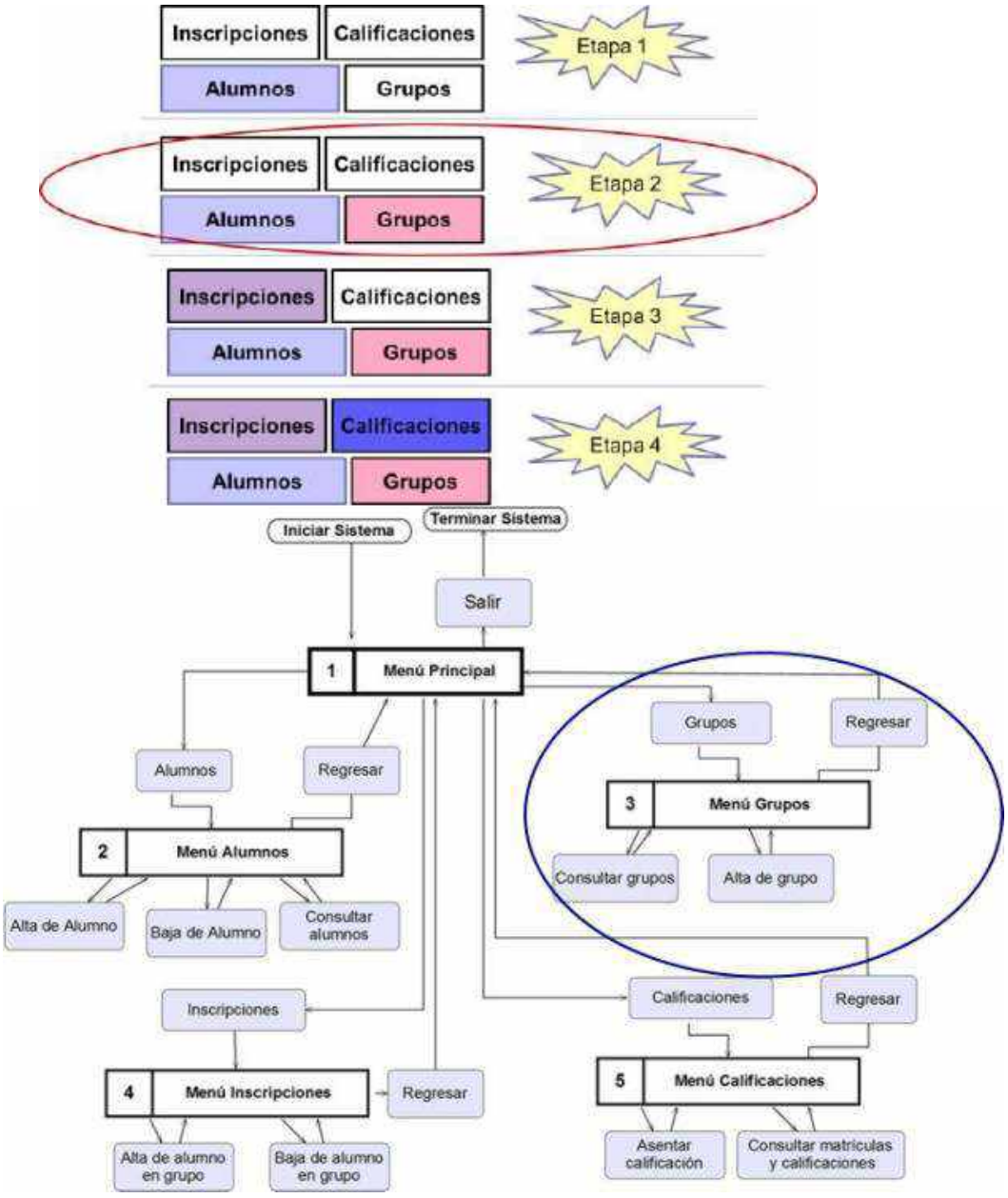


Figura 11-17: En la segunda etapa se desarrolla funcionalidad para el Menú Grupos

11.4.1 Requerimientos a desarrollar en la segunda etapa: *funcionamiento del menú grupos*

R4.- Requerimientos de funcionamiento de la opción alta de grupo

R4.1.- Al seleccionar "Alta de grupo" se solicita al usuario la clave del nuevo grupo, el nombre de la materia y el nombre completo del profesor.

- En un archivo llamado "grupos.txt" se agregará un renglón con los datos proporcionados.
- Se generará un archivo cuyo nombre será: "claveGrupoAlumnos.txt" en el que se irán agregando alumnos posteriormente.
- Se notificará al usuario si el grupo se dio de alta con éxito o no.

R5.- Requerimientos de funcionamiento de la opción consultar grupos

R5.1.- Al seleccionar "Consultar Grupos" se desplegarán:

- La clave
- La materia
- El profesor

De todos los grupos existentes.

11.4.2 Diseño para la segunda etapa

La clase que se necesita para operar con los grupos es `Grupo`. En la Figura 11-18 se describen sus atributos y sus métodos.

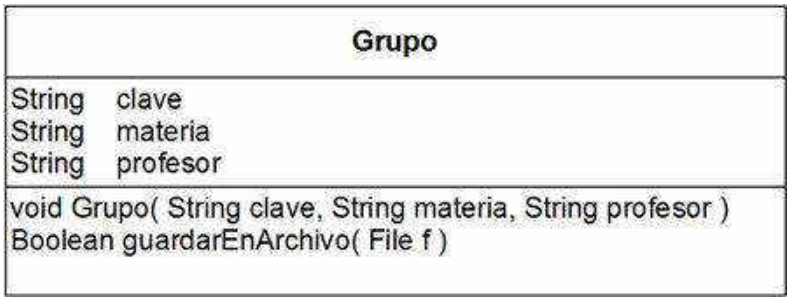


Figura 11-18: La clase `Grupo`

El algoritmo de funcionamiento para ***dar de alta a un grupo*** se ilustra en el diagrama de flujo de la Figura 11-19

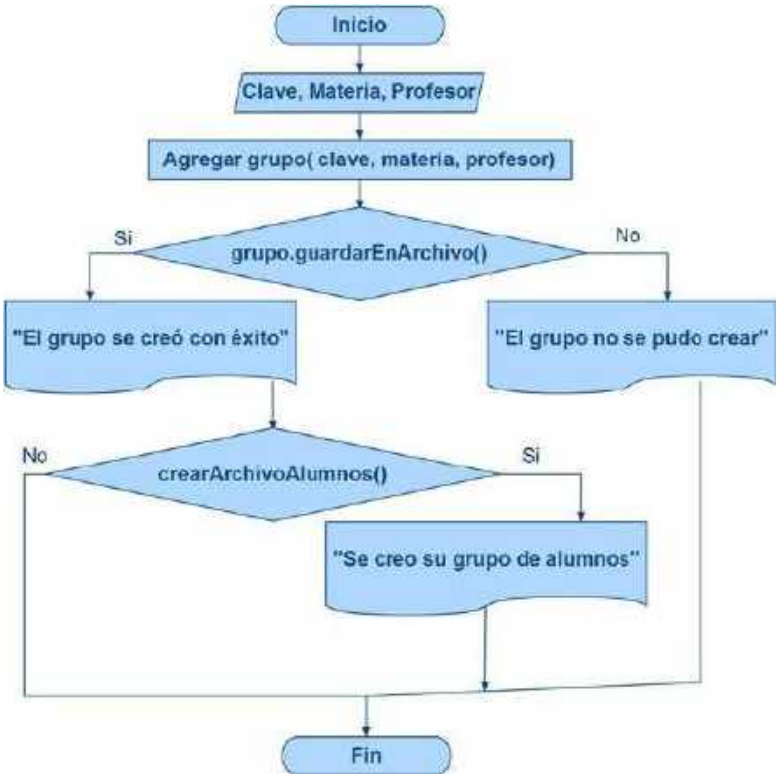


Figura 11-19: Diagrama de flujo del algoritmo para dar de alta a un grupo

11.4.3 Pruebas de validación de la segunda etapa

A continuación se muestran las pruebas de validación del menú grupos, con las opciones: Alta de grupo y Consultar grupo.

I.1.- Se selecciona la opción 2 del menú principal. Luego la opción 1 del menú de grupos. Posteriormente se proporcionan los siguientes datos:

```
Clave: 23-C
Materia: Matematicas
Profesor: Mario Bezares
```

El sistema deberá dar de alta al grupo.

I.2.- Se selecciona la opción 2 del menú de grupos, deberá aparecer el grupo 23-C con la materia `Matemáticas` y el profesor `Mario Bezares`.

I.3.- Repetir las pruebas I.1 y I.2 con cada uno de los siguientes datos

```
27-D, Fisica, Arnulfo Garcia
19-A, Historia, Elena Alcantara
21-B, Etica, Martha Sanchez
```

El archivo “grupos.txt” debe contener:

```
23-C, Matematicas, Mario Bezares
27-D, Fisica, Arnulfo Garcia
19-A, Historia, Elena Alcantara
21-B, Etica, Martha Sanchez
```

Además, en la carpeta del proyecto, se deben haber creado los siguientes archivos:

```
23-CGrupoAlumnos.txt
27-DGrupoAlumnos.txt
19-AGrupoAlumnos.txt
21-BGrupoAlumnos.txt
```

En la Figura 11-20 se ilustra el funcionamiento correcto de las pruebas de aceptación de esta etapa.

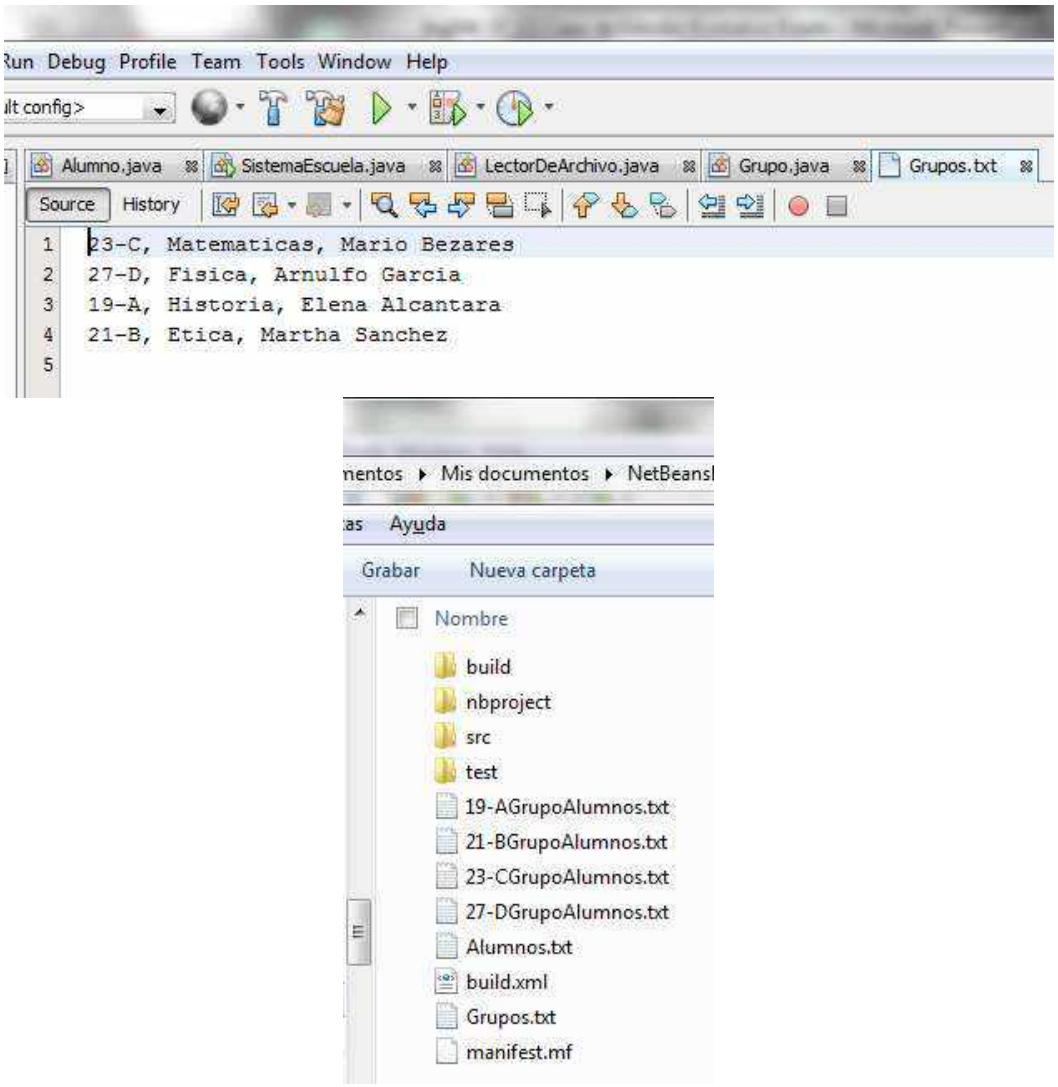


Figura 11-20: Correcto funcionamiento de la funcionalidad de la segunda etapa

11.4.4 Codificación de la segunda etapa

Dejamos al lector la codificación de esta etapa. Los diagramas de flujo de la sección del diseño sirven como guía para elaborar el código. A continuación se proporcionan algunos tips de ayuda.

- Ayuda para la codificación del método `altaGrupo( )`

```
...
   Grupo grupo = new Grupo( clave, materia, profesor );
   // El nombre del archivo que tendrá los alumnos del grupo
   // es la concatenación de la clave con "GrupoAlumnos.txt"
   String nombreArchivo = clave + "GrupoAlumnos.txt";

   if( grupo.guardarEnArchivo( f ) ){
       System.out.println( "El grupo se creó con éxito" );
       // Se crea un archivo para guardar los alumnos del grupo
       File fGrupoAlumnos  = new File( nombreArchivo );
       if ( creaArchivoAlumnos( fGrupoAlumnos ) );
            System.out.println( "Se creó su grupo de alumnos " );
   }
   else
       System.out.println("El grupo no se pudo crear ");
```

- Ayuda para la codificación del método `crearArchivoAlumnos( )`

```
   try{
      // append = true para agregar sin sobreescribir
      FileWriter w = new FileWriter( f, true );
      BufferedWriter bw = new BufferedWriter(w);
      PrintWriter salida = new PrintWriter( bw );
      salida.close();
      bw.close();
      exito = true;
   }catch(IOException e){};
   return exito;
```

11.5 Etapa 3 del desarrollo incremental

En esta tercera etapa desarrollaremos la funcionalidad del menú "Inscripciones", como se ilustra en la Figura 11-21.

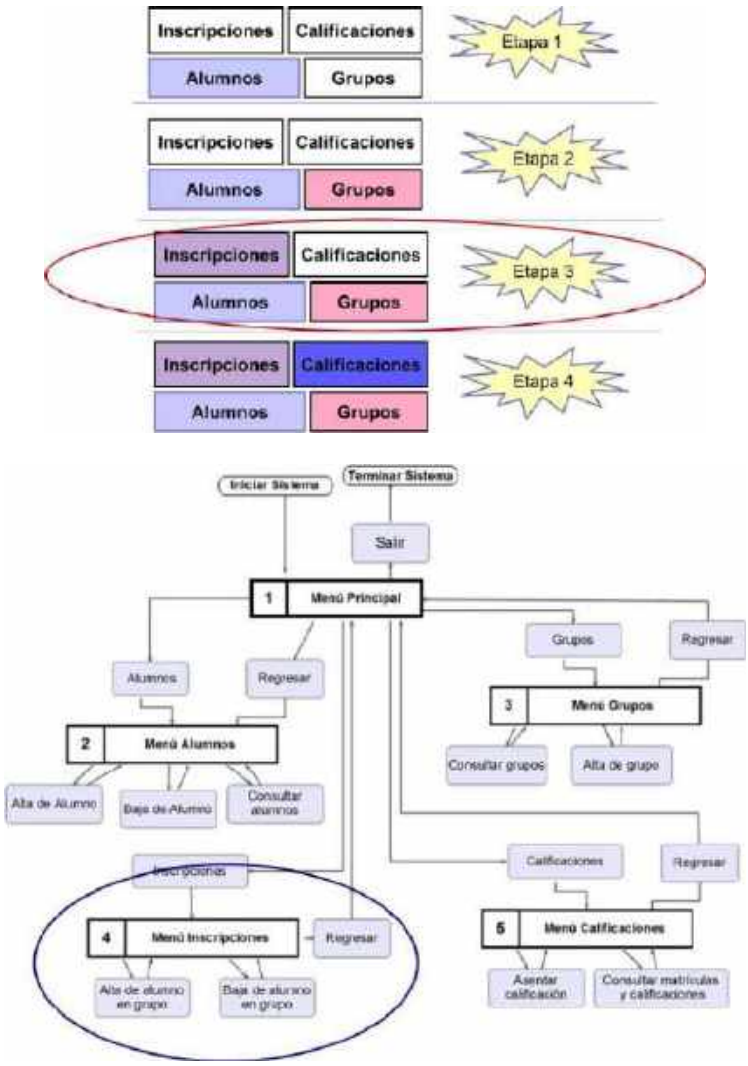


Figura 11-21: En la tercera etapa se desarrolla funcionalidad para el Menú Inscripciones

11.5.1 Requerimientos a desarrollar en la tercera etapa: *funcionamiento del menú inscripciones*

R6.- Requerimientos de funcionamiento de la opción alta de alumno en un grupo

R6.1.- Al seleccionar "Alta de alumno en grupo" se solicita su matrícula, y la clave del grupo al que se desea inscribir, se busca el archivo "claveGrupoAlumnos.txt" y se agrega la matrícula del alumno, con calificación "-1" y la clave del grupo en el archivo.

R6.2.- En caso de que no se encuentre un grupo con la clave proporcionada se le indicará al usuario.

R6.3.- En caso de que no se encuentre la matrícula se indicará al usuario.

R7.- Requerimientos de funcionamiento de la opción baja de alumno en un grupo

R7.1.- Al seleccionar "Baja de alumno en grupo" se solicita su matrícula, y la clave del grupo del que se desea dar de baja, se busca el archivo "claveGrupoAlumnos.txt" y se borra en ese archivo el renglón correspondiente a la matrícula dada.

R7.2.- En caso de que no se encuentre un grupo con la clave proporcionada se le indicará al usuario.

R7.3.- En caso de que no se encuentre la matrícula se indicará al usuario.

11.5.2 Diseño para la tercera etapa

La clase que se necesita para trabajar con las inscripciones es `Inscripcion`. En la Figura 11-22 se describen sus atributos y sus métodos.

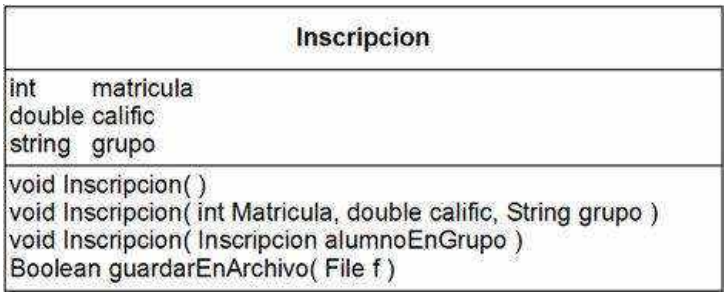


Figura 11-22: La clase `Inscripcion`

El algoritmo de funcionamiento para ***dar de alta a un alumno en un grupo*** se ilustra en el diagrama de flujo de la Figura 11-23.

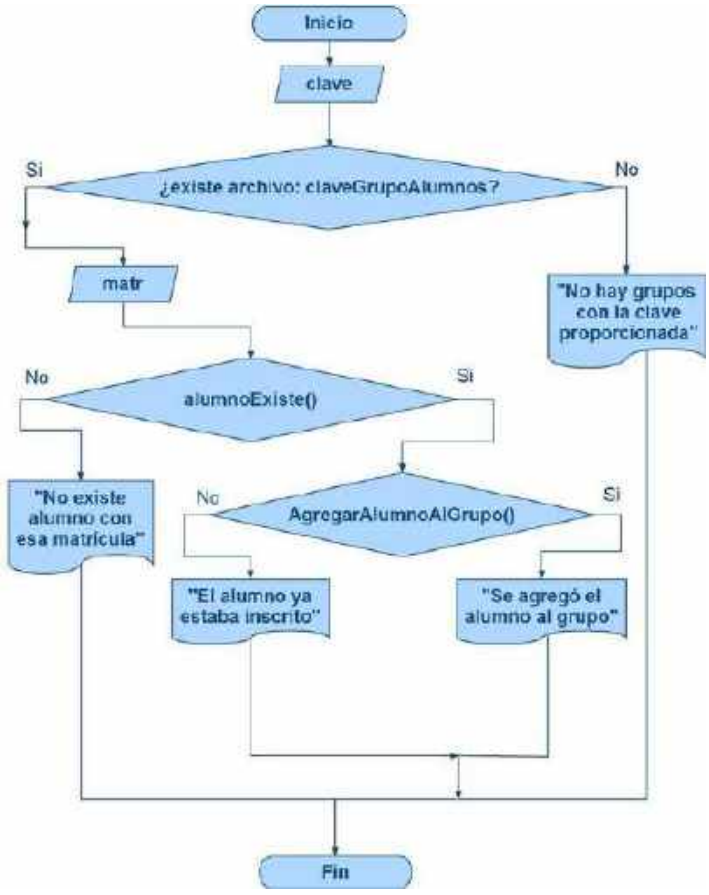


Figura 11-23: Diagrama de flujo del algoritmo para dar de alta a un alumno en un grupo

El algoritmo de funcionamiento para ***dar de baja a un alumno en un grupo*** se ilustra en el diagrama de flujo de la Figura 11-24.

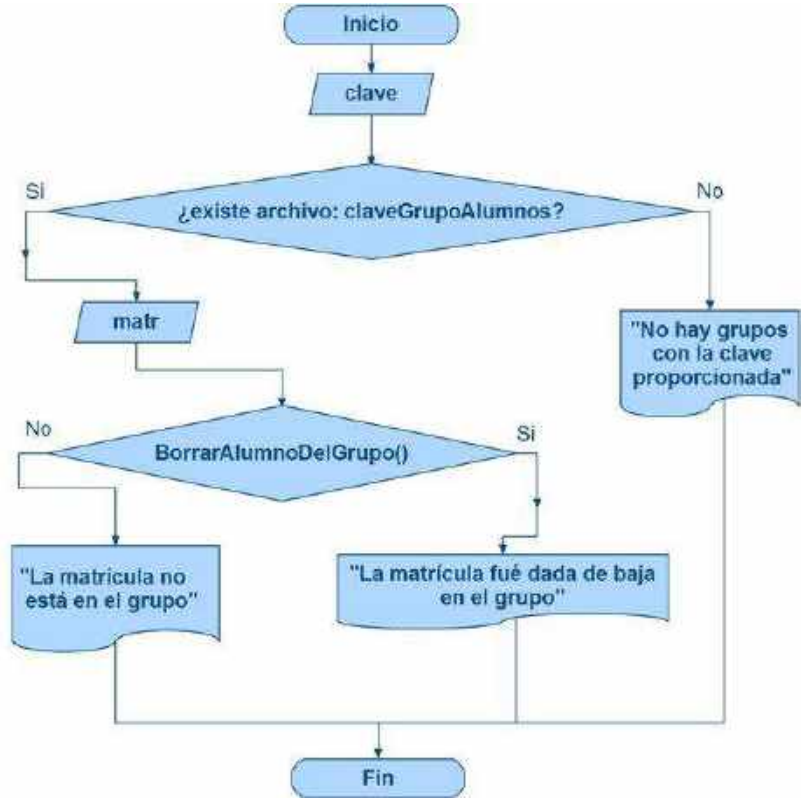


Figura 11-24: Diagrama de flujo del algoritmo para dar de baja a un alumno en un grupo

Como se aprecia en el algoritmo de la Figura 11-24, se requiere diseñar un procedimiento para borrar a un alumno de un grupo, éste procedimiento se describe con el algoritmo de la Figura 11-25.

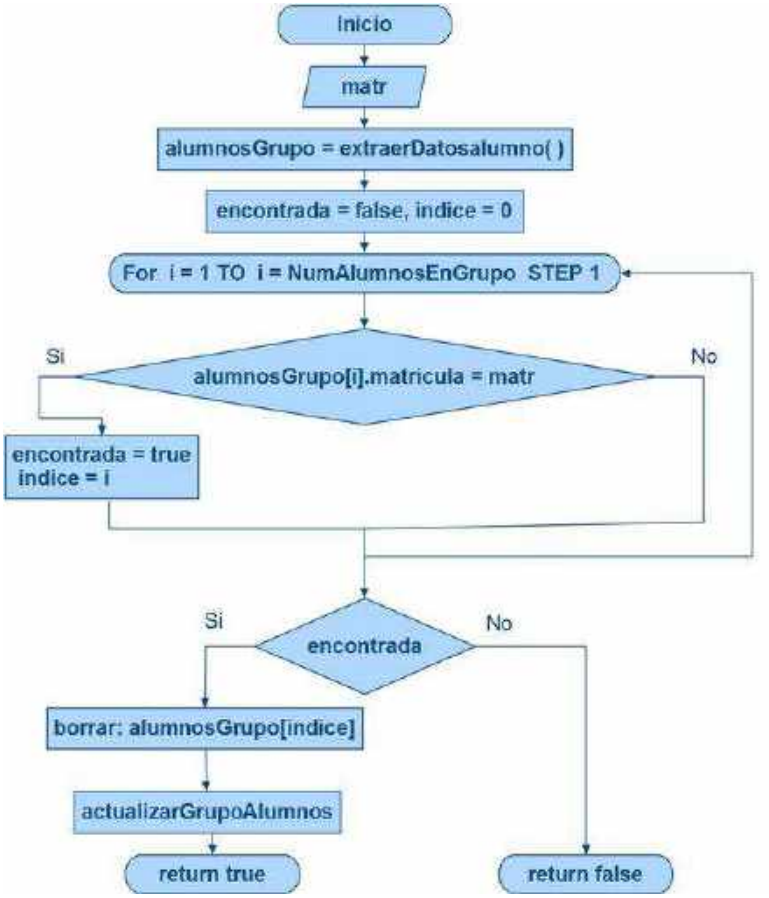


Figura 11-25: Diagrama de flujo del algoritmo para borrar a un alumno de un grupo

11.5.3 Codificación y pruebas de validación de la tercera etapa

Los diagramas de flujo de la sección del diseño sirven como guía para elaborar el código. A continuación, se proporcionan algunos de los métodos ya codificados y tips de ayuda para codificar los demás. Dividiremos esta etapa en dos bloques, en el primero se codificará la funcionalidad de dar de alta a un alumno en un grupo y en el segundo la de darlo de baja.

11.5.3.1 Codificación del primer bloque de la tercera etapa

En este primer bloque codificaremos la funcionalidad para dar de alta un alumno en un grupo.

- Ayuda para la codificación del método `altaAlumnoEnGrupo( )`

```
private static void altaAlumnoEnGrupo() {
  ...
   if( existeGrupoAlumnos( nombreArchivo )){
      System.out.println( "Matricula del alumno? ");
      matr = input.nextInt();
      if( alumnoExiste( matr )){
```

```
          if( agregaAlumnoAlGrupo( nombreArchivo, matr ))
            System.out.println("El alumno quedó inscrito en el grupo");
          else
            System.out.println("...???...");
        }
        else
          System.out.println("...???...");
     }
     else
        System.out.println( "...?..." );

}
```

El código del método `existeGrupoAlumnos()` es el siguiente:

```
private static boolean existeGrupoAlumnos( String nombreArchivo ) {
   File archivo = null;
   FileReader fr = null;
   BufferedReader br = null;
   Boolean existe = false;
   try {
      archivo = new File ( nombreArchivo );
      fr = new FileReader (archivo);
      br = new BufferedReader(fr);
      existe = true; // Se pudo abrir el archivo
   }
   catch(Exception e){
      //e.printStackTrace();
   }finally{
     // se cierra el archivo tanto para el caso normal
     //  como para la excepción
     try{
        if( null != fr )
           fr.close();
     }catch (Exception e2){
      e2.printStackTrace();
     }
   }
   return existe;
}
```

No imprimiremos el stackTrace cuando no existe el archivo

El código del método `alumnoExiste()` es el siguiente:

```
private static boolean alumnoExiste(int matr) {
    ArrayList<String> alumnosString = new ArrayList<String>();
    ArrayList<Alumno> alumnos = new ArrayList<Alumno>();
    LectorDeArchivo lectorDeArchivo = new LectorDeArchivo("Alumnos.txt");
    alumnosString = lectorDeArchivo.LeerArchivo();
    alumnos = extraerDatosAlumno( alumnosString );

    Boolean encontrada = false;
    for( int i=0; i < alumnos.size() ; i++ ){
        if( alumnos.get(i).getMatricula() == matr )
            encontrada = true;
    }
    if( encontrada )
        return true;
    else
        return false;
```

```
}
```

Completar el código del método `agregaAlumnoAlGrupo()`:

```
private static Boolean agregaAlumnoAlGrupo( String nombreArchivo,
                                            int matr,
                                            String clave ) {
  File fGrupoAlumnos  = new File( nombreArchivo );
  Boolean exito = false;
  ArrayList<String> alumnosEnGrupoStr = new ArrayList<String>();
  ArrayList<Inscripcion> alumnosEnGrupo = new ArrayList<Inscripcion>();
  LectorDeArchivo lectorDeArchivo = new LectorDeArchivo( nombreArchivo );
  alumnosEnGrupoStr = lectorDeArchivo.LeerArchivo();
  alumnosEnGrupo = extraerDatosAlumnoEnGrupo( alumnosEnGrupoStr );

  Inscripcion inscripcion = new Inscripcion( matr, -1, clave );

  Boolean yaInscrito = false;
  for( . . .){
     . . .
  }
  if( yaInscrito ){
      return false;
  }
  else
    {
      exito = inscripcion.guardarEnArchivo( fGrupoAlumnos );
      return exito;
    }
}
```

11.5.3.2 *Pruebas de validación para el primer bloque de la tercera etapa: alta de alumno en un grupo*

I.1.- Se selecciona la opción 3 del menú principal. Luego la opción 1 del menú de inscripciones. Se darán los siguientes datos:

```
Clave: 23-C
Matricula: 5
```

El sistema deberá dar de alta al alumno con matrícula 5 en el grupo 23-C. Esto se verifica consultando el archivo 23-CGrupoAlumnos.txt el cual debe contener:

```
Alumnos del grupo
   5, -1
```

I.2.- Se repite I.1 con los datos:

```
Clave: 32-A
```

El sistema deberá decir que no existe grupo con esa clave.

I.3.- Repetir la pruebas I.1 y con los datos

```
Clave: 23-C
Matricula: 8
```

El sistema deberá indicar que no existe un alumno con esa matrícula.

I.4.- Se repite I.1 con los siguientes datos:

```
Clave: 23-C
Matricula: 1
```

El sistema deberá dar de alta al alumno con matrícula 1 en el grupo 23-C. Esto se verifica consultando el archivo 23-CGrupoAlumnos.txt el cual debe contener:

```
Alumnos del grupo
   5, -1, 23-C
   1, -1, 23-C
```

I.5.- Se repite I.1 inscribiendo al alumno con matrícula 3 en los grupos:

```
19-A, 21-B  y 23-C
```

Verificar que esté la matrícula en los archivos correspondientes.

I.6.- Se repite I.1 inscribiendo al alumno con matrícula 2 en los grupos:

```
19-A, 21-B, 23-C y 27-D
```

Verificar que esté la matrícula en los archivos correspondientes.

I.7.- Se repite I.1 inscribiendo al alumno con matrícula 2 en el grupo:

```
19-A
```

El sistema deberá indicar que el alumno ya estaba inscrito.

En la Figura 11-26 se ilustra el contenido de cada uno de los archivos de alumnos del grupo después de correr las pruebas de aceptación.

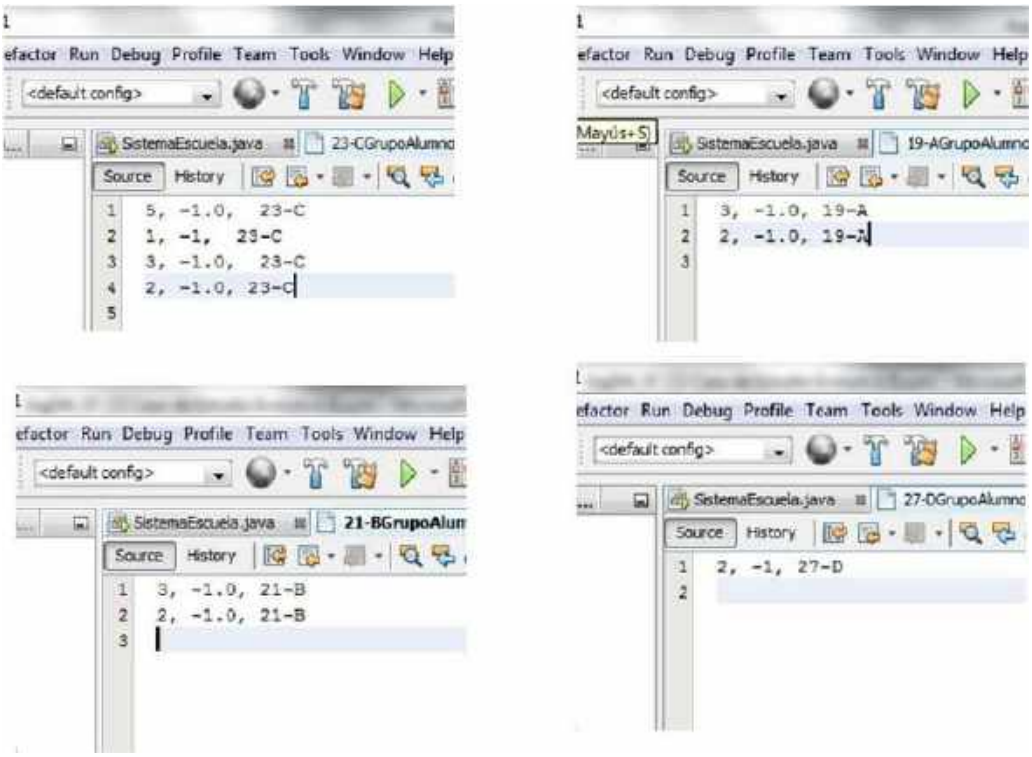


Figura 11-26: Contenido de los archivos después de correr las pruebas de aceptación

11.5.3.3 Codificación del segundo bloque de la tercera etapa

En este segundo bloque se deja al lector la codificación de la funcionalidad para dar de baja a un alumno de un grupo. Los diagramas de flujo de la sección del diseño sirven como guía para elaborar el código. Como una ayuda, se proporciona el código del método `actualizarGrupoAlumnos()`:

```
private static void actualizarGrupoAlumnos(
                                String                  nombreGrupo,
                                ArrayList<Inscripcion> grupoAlumnos) {

   File file = new File( nombreGrupo );
   File archivo = file;
   file.delete();  // Se borra el archivo anterior
   Inscripcion alumnoEnGrupo = new Inscripcion();

   for( int i=0; i < grupoAlumnos.size(); i++ ){
       alumnoEnGrupo = new Inscripcion( grupoAlumnos.get(i) );
       // los datos actualizados quedan en "claveGrupoAlumos.txt"
       alumnoEnGrupo.guardarEnArchivo( archivo );
   }
}
```

11.5.3.4 *Pruebas de validación para el segundo bloque de la tercera etapa: baja de alumno en un grupo*

I.1.- Se selecciona la opción 3 del menú principal. Luego la opción 2 del menú de inscripciones. Se darán los siguientes datos:

```
Clave: 23-C
Matricula: 2
```

El sistema deberá dar de baja al alumno con matrícula 2 en el grupo 23-C. Esto se verifica consultando el archivo 23-CGrupoAlumnos.txt

I.2.- Se repite I.1 pero con los datos

```
Clave: 23-C
Matricula: 8
```

El sistema deberá decir que no existe un alumno con esta matrícula.

I.3.- Se repite I.1 dando como clave de grupo:

```
Clave: 51-F
```

El sistema deberá decir que no hay grupos con la clave proporcionada.

En la Figura 11-27 se ilustra el funcionamiento correcto de las pruebas de aceptación del segundo bloque de la tercera etapa.

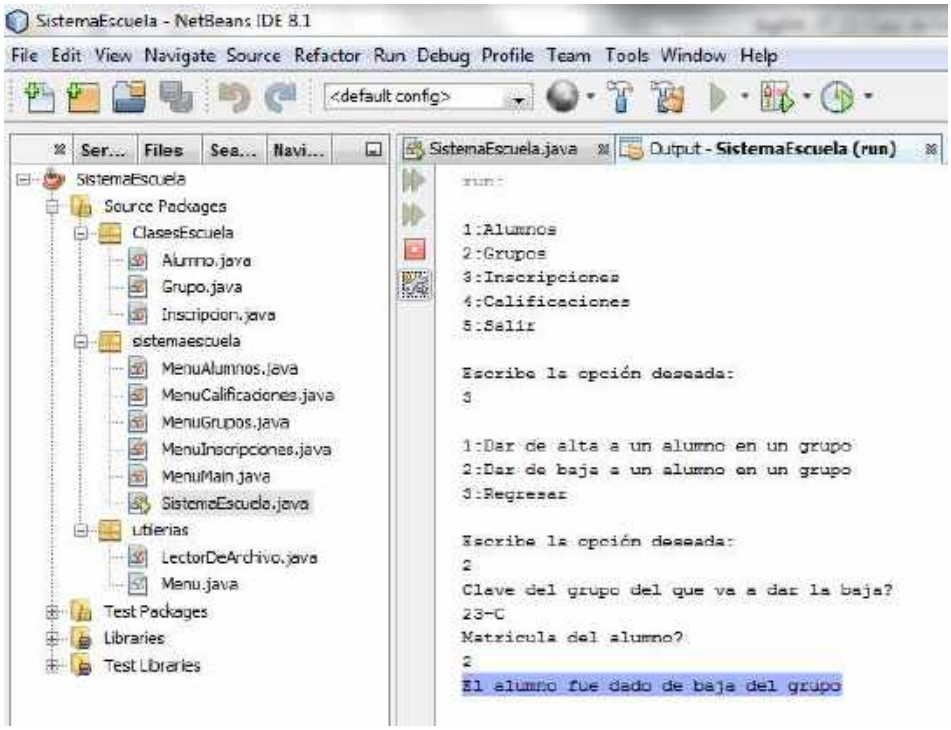


Figura 11-27: Correcto funcionamiento de la funcionalidad: dar de baja a un alumno de un grupo

11.6 Etapa 4 del desarrollo incremental

En esta cuarta etapa desarrollaremos la funcionalidad del menú "Calificaciones", como se ilustra en la Figura 11-28.

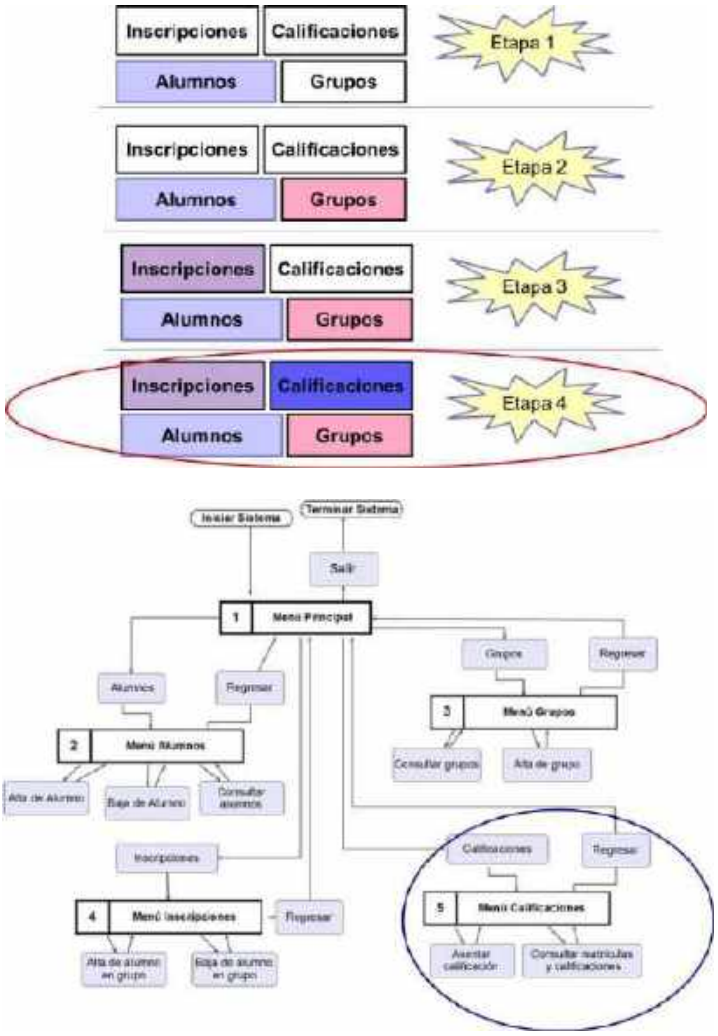


Figura 11-28: En la cuarta etapa se desarrolla funcionalidad para el Menú Calificaciones

11.6.1 Requerimientos a desarrollar en la cuarta etapa: *funcionamiento del menú calificaciones*

R8.- Requerimientos de funcionamiento de la opción asentar calificación

R8.1.- Al seleccionar “Asentar calificación” se solicita al usuario la clave del grupo, la matrícula y la calificación. Se busca el archivo “claveGrupo.txt” y se modifica la calificación correspondiente a la matrícula proporcionada.

R8.2.- Se notificará al usuario cuando se haya asentado la calificación con éxito.

R8.3.- En caso de que no se localice el grupo o la matrícula proporcionados, se indicará al usuario.

R9.- Requerimientos de funcionamiento de la opción consultar matrículas y calificaciones

R9.1.- Al seleccionar "Consultar matrículas y calificaciones" se solicita al usuario la clave del grupo. Se busca el archivo "claveGrupo.txt" y se despliegan las matrículas y sus calificaciones asociadas.

R9.2.- En caso de que no se encuentre un grupo con la clave proporcionada se le indicará al usuario.

11.6.2 Diseño para la cuarta etapa

En esta etapa trabajaremos también con la clase `Inscripcion`. El algoritmo de funcionamiento para ***consultar matrículas y calificaciones de un grupo*** se ilustra en el diagrama de flujo de la Figura 11-29.

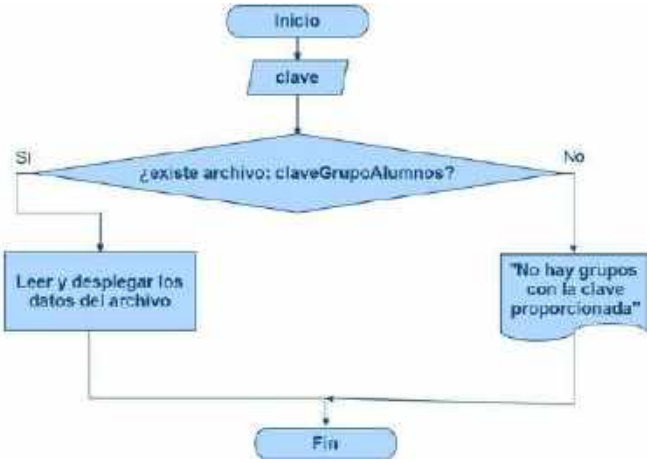


Figura 11-29: Diagrama de flujo del algoritmo para consultar matrículas y calificaciones de un grupo

El diagrama de flujo para calificar a un alumno se muestra en la Figura 11-30.

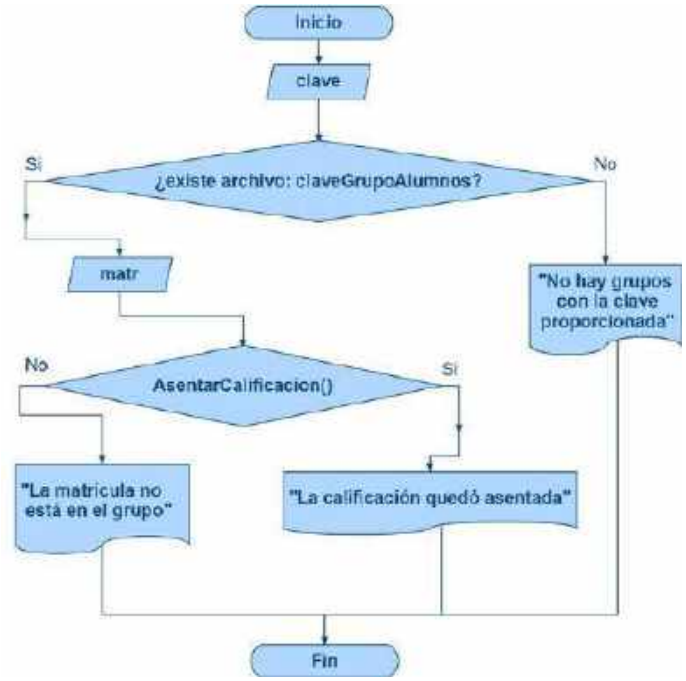


Figura 11-30: Diagrama de flujo para calificar a un alumno

Como se observa en el diagrama anterior, se requiere de un mecanismo para asentar la calificación, el cual se ilustra en la Figura 11-31.

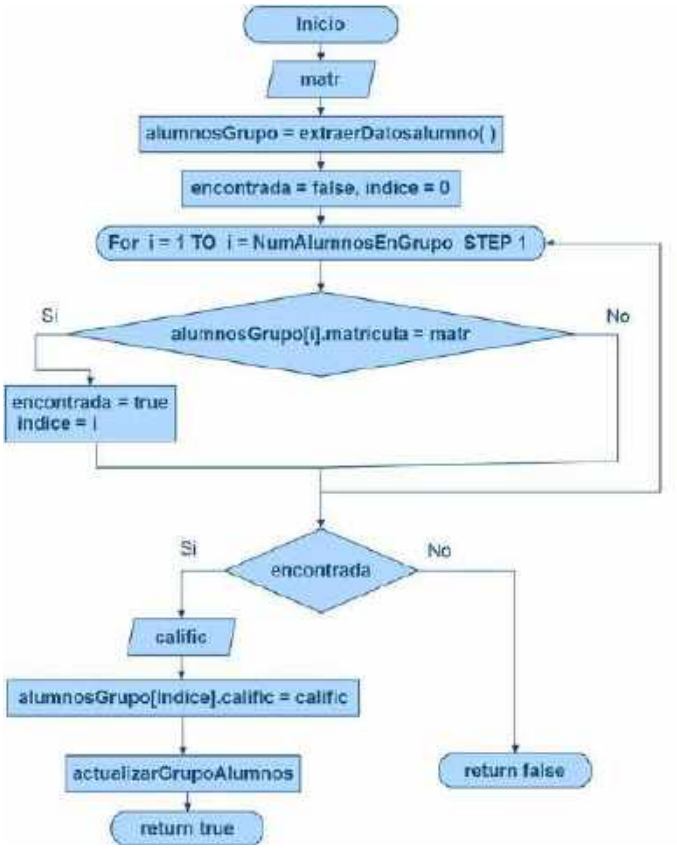


Figura 11-31: Diagrama de flujo del mecanismo para asentar una calificación

11.6.3 Codificación y pruebas de validación de la cuarta etapa

Dejamos al lector la codificación de esta etapa. Los diagramas de flujo de la sección del diseño sirven como guía para elaborar el código. A continuación se proporciona una ayuda para codificar el método `asentarCalificacion()`.

```
private static boolean asentarCalificacion(String  nombreArchivo,
                                           ArrayList<String> grupoAlumnStr,
                                           int         matricula ) {
  ArrayList<Inscripcion> grupoAlumnos    =  ArrayList<Inscripcion>();
  ArrayList<Inscripcion> grupoAlumnosAct = new ArrayList<Inscripcion>();

  grupoAlumnos = extraerDatosAlumnoEnGrupo( grupoAlumnStr );
  Inscripcion [] arregloAlumnos =
                        new Inscripcion[ grupoAlumnos.size() ];
  Boolean encontrada = false;
  int indice = 0;
  Scanner   input = new Scanner( System.in );
  double   calific;
 // Poner el contenido de grupoAlumnos en arregloAlumnos
 // Buscar la matrícula y encender encontrada si se encuentra
 // Si se encontró la matrícula:
 //   - Poner la calificación a la matrícula proporcionada
 //   - Actualizar el arraylist de grupoAlumnos
 //   - Actualizar el archivo .txt correspondiente
```

11.6.3.1 *Pruebas de validación para la funcionalidad de la cuarta etapa: menú calificaciones*

I.- Asentar una calificación

I.1.- Se selecciona la opción 4 del menú principal. Luego la opción 1 del menú calificaciones. Se darán los siguientes datos:

```
Clave: 23-C
Matricula: 1
Calificación: 8.4
```

El sistema deberá asentar la calificación a la matrícula proporcionada.

I.2.- Repetir I.1 proporcionando una matrícula que no esté dada de alta en el grupo, por ejemplo:

```
Clave: 19-A
Matricula: 1
```

El sistema deberá indicar que la matrícula no está en el grupo.

I.3.- Repetir I.1 proporcionando la clave de un grupo que no exista, por ejemplo:

```
Clave: 30-A
```

El sistema deberá indicar que no existe un grupo con esa clave.

En la Figura 11-32 se ilustra un ejemplo de prueba de validación cuando el intento de asentar la calificación es exitoso, es decir, cuando *existe el grupo y la matrícula* proporcionados.

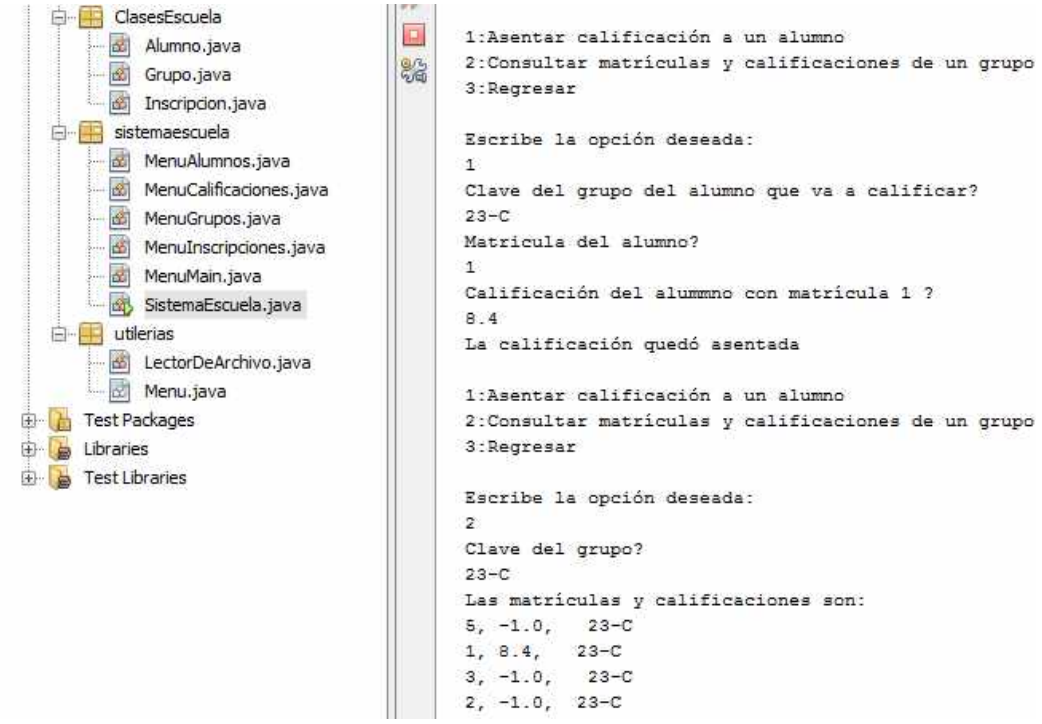
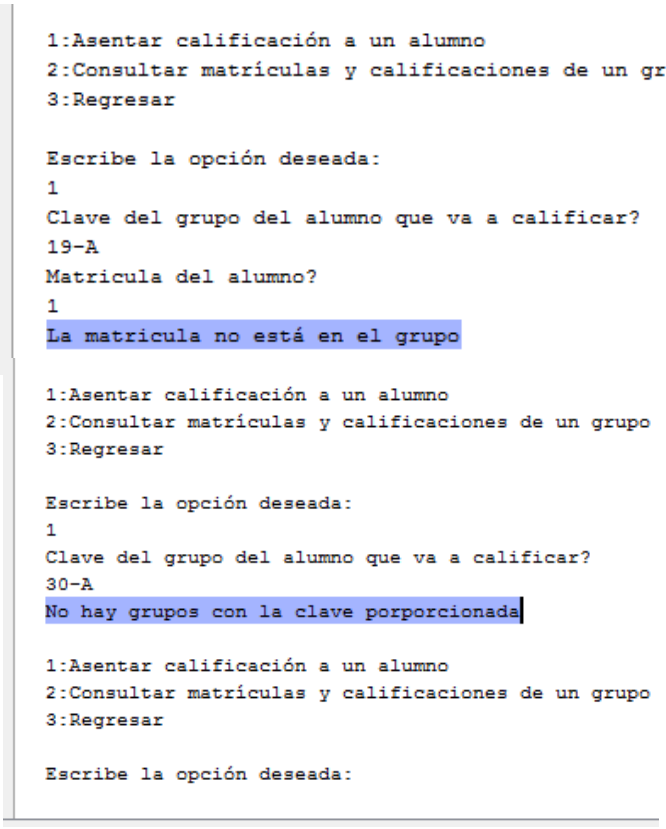


Figura 11-32: Asentar la calificación cuando existe el grupo y la matrícula proporcionados

En la Figura 11-33 se ilustra un ejemplo de prueba de validación en la que el intento de asentar la calificación no es exitoso, en el caso a) *existe el grupo y pero no la matrícula* proporcionada y en el caso b) *no existe el grupo* proporcionado.



a) Existe el grupo pero no la matrícula **b) No existe el grupo**

Figura 11-33: Prueba cuando el intento de asentar calificación no es exitoso

II.- Consultar matrículas y calificaciones de un grupo

I.1.- Se selecciona la opción 4 del menú principal. Luego la opción 2 del menú de calificaciones. Se proporciona la clave de un grupo existente (23-C, 27-D, 19-A o 21-B)

El sistema deberá desplegar los datos del archivo correspondiente (23-CGrupoAlumnos.txt, …..)

12 Modelos de Reuso y modelos Híbridos

12.1 Objetivos específicos del capítulo

- Conocer los principios del proceso del desarrollo basado en componentes.
- Conocer las características y limitaciones del desarrollo basado en componentes.
- Conocer los principios del Proceso de Desarrollo Unificado.
- Conocer las características y limitaciones del Proceso de Desarrollo Unificado.

12.2 Los modelos de Reusabilidad

[9]La reusabilidad (reuso) tiene como objetivo utilizar algunos elementos ya entregados del producto de software, tales como elementos de la especificación de requerimientos, del diseño, del código o de cualquier producto comercialmente disponible. Va desde la reutilización de clases y métodos en las bibliotecas hasta la reutilización de sistemas de aplicación completos. La reusabilidad en el desarrollo de sistemas se basa en la existencia

[9] http://aureliedeville.be/photos/photomontages/archi.jpg

de componentes reutilizables, y tiene como finalidad usar de nuevo ideas, arquitecturas, diseños o código de una aplicación para construir otras. De aquí que dos aspectos clave a considerar en la calidad de un sistema de software a desarrollar, sean su reusabilidad y su extensibilidad.

El proceso de desarrollo de un sistema con reuso se enfoca en integrar los componentes reutilizables en el sistema en lugar de desarrollarlos desde cero. A la versión moderna de reuso se le llama ***Desarrollo basado en componentes***.

12.2.1 El Desarrollo Basado en Componentes

El desarrollo basado en componentes afronta la complejidad de los sistemas y los sistemas a gran escala por medio del ensamblado de componentes.

El desarrollo basado en componentes se divide en dos ramas: el *desarrollo de componentes* y el *desarrollo de sistemas utilizando componentes* [Crnkovic y Larsson, 2005], los cuales se explican a continuación.

Desarrollo de componentes.- Es el que se enfoca en la construcción de componentes reutilizables, pensando en que van *a* ***ser parte de*** un sistema. Las etapas del *desarrollo de componentes*, según [Christianson et al., 2002], son: Análisis y definición de requerimientos de los componentes, Diseño para la reutilización, Especificación de los componentes, Implementación, Verificación y validación, Mantenimiento y soporte técnico. En la Figura 12-1 se ilustran estas etapas.

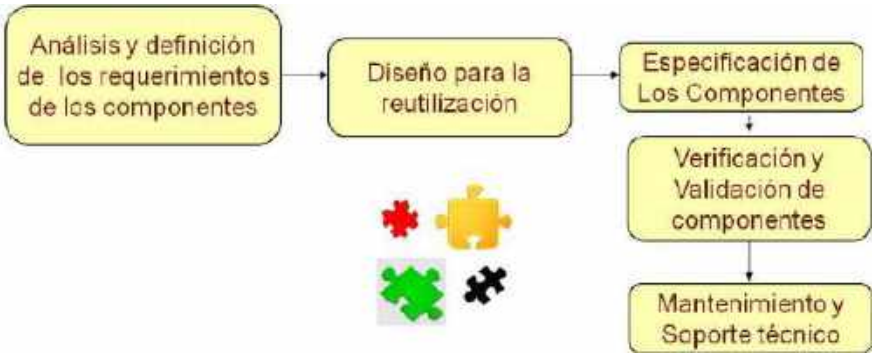
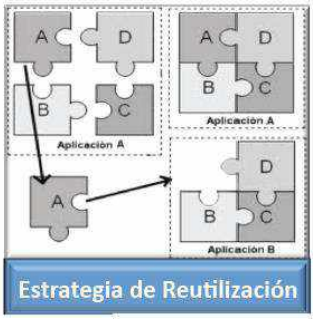


Figura 12-1: Etapas del desarrollo de componentes

Desarrollo de sistemas utilizando componentes.- También llamado ***desarrollo con reutilización de componentes***. Se enfoca en identificar y ***adaptar componentes*** para ***ser reutilizados*** en la construcción de un sistema. Las etapas de especificación de requerimientos y de validación son comparables con las de los otros modelos de desarrollo, sin embargo, las etapas intermedias en el proceso orientado a la reutilización son diferentes. Las etapas del *desarrollo del sistema utilizando componentes*, según Sommerville (2007), son:



[10]*Análisis de componentes*. Consiste en encontrar componentes que sirvan para implementar la especificación de requerimientos. En general los componentes que se utilizan solo proporcionan parte de la funcionalidad requerida, por lo que durante el análisis se determina en dónde es necesario modificarlos para que éstos sean útiles.

Modificación de requerimientos. Con la información que se tiene de los componentes ya identificados, se analizan los requerimientos. Si es posible, se modifican los requerimientos para que concuerden con los componentes disponibles. Si las modificaciones no son posibles entonces se lleva a cabo nuevamente el análisis de componentes para buscar soluciones alternativas.

Diseño del sistema con reutilización. Se diseña teniendo en cuenta los componentes que se reutilizan y los componentes que serán completamente nuevos. Los componentes se incorporan al sistema por medio de un marco de trabajo (framework), que es el contexto para juntar todas las piezas.

Desarrollo e integración. El software que no se tiene disponible y que no se puede adquirir externamente se desarrolla integrando los componentes reutilizables disponibles. En este modelo, la integración de los sistemas es parte del desarrollo en lugar de una actividad separada.

En la Figura 12-2 se ilustran las etapas del desarrollo de sistemas con reutilización de componentes.

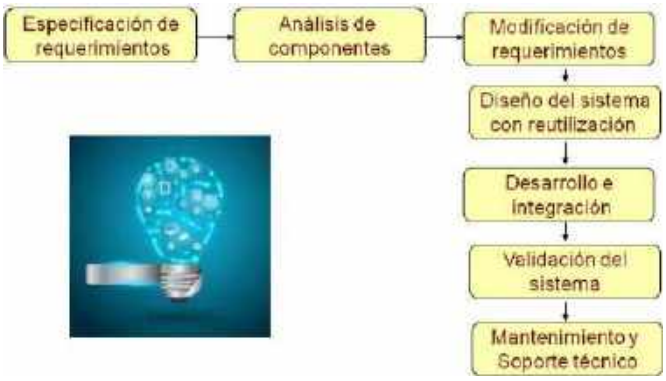


Figura 12-2: Etapas del desarrollo de sistemas usando componentes[11]

[10] https://www.freepik.es/fotos-vectores-gratis/rompecabezas-forma

[11] http://geekchicpro.com/wp-content/uploads/st/stock-illustration-vector-light-bulb-with-technology.jpg

12.2.2 Ejemplos de desarrollo basado en componentes

A continuación presentamos dos ejemplos de tipos de desarrollo basado en componentes, el de líneas de productos y el COTS.

- ***Líneas de productos (Product Line)***.- Se construyen *familias de productos* (llamadas líneas de productos) que comparten la misma arquitectura y se construyen a partir de componentes comunes. Cada línea de producto pertenece a un dominio de aplicación específico o a una estrategia de mercado. El objetivo es vender casi el mismo sistema a muchos clientes con costos mínimos. Ejemplos: una aplicación para diferentes Smartphone, motor de un videojuego para diferentes plataformas.

- ***COTS-based software development***.- **(COTS: Commercial On The Shell) .-** Un producto COTS es un sistema de software de uso general, que incluye muchas características y funcionalidades que pueden adaptarse a las necesidades de diferentes clientes sin tener que cambiar el código fuente. Ejemplos: Paquetes ERP (Enterprice Resource Planning) para la gestión de nómina, facturas, ventas, etc., aplicaciones ofimáticas (procesador de texto, hoja de cálculo, correo electrónico,..), paquetes de diseño asistido por computadora.

12.2.3 Ventajas y desventajas del reuso del software

El reuso del software tiene la ventaja obvia de reducir la cantidad de software a desarrollarse, por lo que se reducen los costos y los riesgos. Sin embargo, si se cambian mucho los requerimientos para adaptarse al sistema anterior, es posible que el nuevo sistema no cumpla las necesidades reales de los usuarios. Además, si las nuevas versiones de los componentes reutilizables no están bajo el control de la organización que los utiliza, se pierde el control sobre la evolución del sistema.

12.3 El Proceso Unificado

12.3.1 Introducción

El Proceso Unificado (RUP: Rational Unified Process) se concibió para el diseño orientado a objetos. Sin embargo, puede usarse para el paradigma estructurado. El RUP es un modelo híbrido porque reúne elementos de todos los modelos de procesos genéricos (cascada, evolutivo y reuso). Sus características predominantes son el desarrollo iterativo, el desarrollo dirigido por riesgos y el desarrollo incremental del sistema. El Proceso Unificado visualiza al sistema desde tres perspectivas:

1.- *Perspectiva dinámica*: son las **fases** del modelo sobre el tiempo.

2.- *Perspectiva estática*: son las actividades del proceso y se les llama **flujos de trabajo**.

3.- *Perspectiva práctica*: sugiere buenas prácticas a utilizar durante el desarrollo del sistema.

Perspectiva dinámica.- Se dice que las fases son sobre el tiempo porque se aplican de forma secuencial, éstas son: Inicio, Elaboración, Construcción y Transición son iterativas (ver Figura 12-3). Cada una de estas fases es iterativa, es decir, pueden constar de una o más iteraciones.

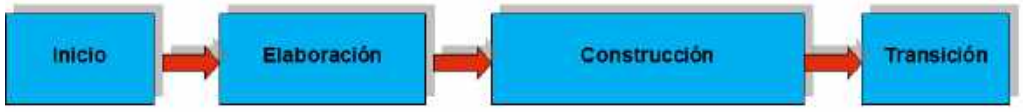


Figura 12-3: Las fases del Proceso Unificado

Las características de la fase de *inicio* son las siguientes:

- Se establece la razón de ser del proyecto y se determina su alcance.
- Se adquiere una visión aproximada del proyecto en términos de los principales requerimientos.
- Se desarrolla un primer análisis del negocio.
- Se efectúan las primeras estimaciones, aún no precisas, del proyecto.
- Ésta no es precisamente una fase de recolección de requerimientos, sino una especie de fase de viabilidad.

A continuación, se mencionan las características de la fase de *elaboración*:

- Recolección de requerimientos más detallados.
- Se realiza análisis y diseño de alto nivel con el objetivo de definir la arquitectura base del sistema.

- Se crea el plan de construcción.
- Se refina la visión que se tiene del proyecto.
- Se identifican más requerimientos.
- Se determina de forma más refinada el alcance del proyecto.
- Análisis y mitigación de los principales riesgos.

Las características de la fase de *construcción* son las siguientes:

- Consta de varias iteraciones.
- En cada iteración se construye software que satisface un subconjunto de los requerimientos del proyecto.
- Los requerimientos a implementar son aquellos de menor riesgo y de fácil implementación.
- El software construido en cada iteración es software de calidad probado e integrado.

A continuación, se mencionan las características de la fase de *transición*:

- Pruebas beta.
- Refinamiento del desempeño del sistema.
- Entrenamiento del usuario.
- Lanzamiento del sistema para su puesta en producción.

En el cuadro de la Figura 12-4 se hace una síntesis con la descripción cada una de las fases del proceso unificado.

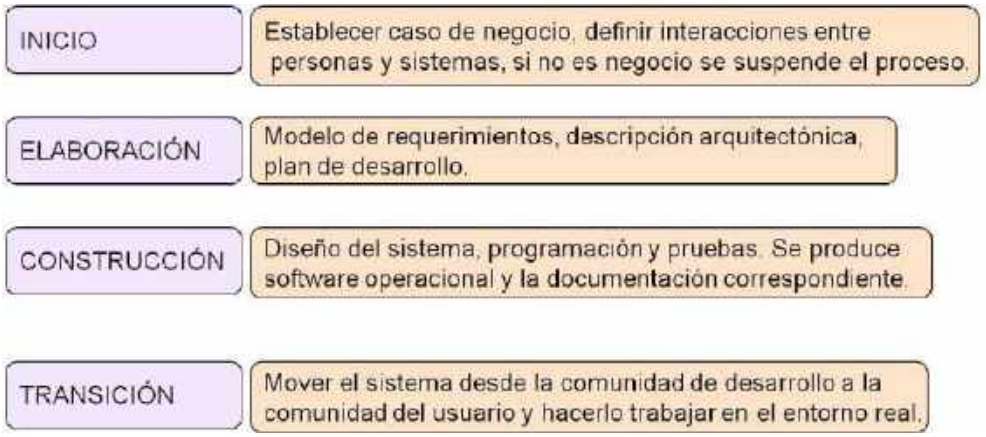


Figura 12-4: Descripción de las Fases del Proceso Unificado

Perspectiva estática.- Los ***flujos de trabajo*** son las actividades que se llevan a cabo durante el proceso, y se ilustran en la tabla de la Figura 12-5. Éstos son similares a las etapas del modelo en cascada, pero además se toman en cuenta otros aspectos.

Flujo de trabajo	*Descripción*
Modelado del negocio	Los procesos del negocio se modelan utilizando casos de uso de negocio
Requerimientos	Se definen los actores que interactúan con el sistema y se desarrollan casos de uso para modelar los requerimientos del sistema
Análisis y diseño	Se crea y documenta un modelo de diseño utilizando modelos arquitectónicos, modelos de componentes, modelos de objetos y modelos de secuencias.
Implementación	Los componentes del sistema se estructuran y se implantan en subsistemas. La generación automática de código de los modelos del diseño ayuda a acelerar este proceso.
Pruebas	Son un proceso iterativo que se lleva a cabo conjuntamente con la implementación. Cuando se termina la implementación, se deben hacer las pruebas del sistema.
Puesta en marcha (despliegue)	Se crea un "release" del producto, se distribuye a los usuarios y se instala en su lugar de trabajo.
Gestión de configuración y cambios	Este flujo de trabajo de soporte gestiona los cambios del sistema.
Gestión del proyecto	Este flujo de trabajo de soporte gestiona el desarrollo del sistema
Entorno	Este flujo de trabajo se refiere a hacer herramientas software apropiadas disponibles para los equipos de desarrollo de software.

Figura 12-5: Los flujos de trabajo del Proceso Unificado

Las *fases son dinámicas* y tienen objetivos. Los *flujos de trabajo son estáticos* y son actividades técnicas que no están asociadas con fases únicas, sino que pueden utilizarse durante el desarrollo para alcanzar los objetivos de cada fase. En la Figura 12-6 se ilustra la combinación de las fases del Proceso Unificado con los flujos de trabajo. Como se puede observar, la intensidad de los flujos de trabajo depende de la fase del proceso, por ejemplo, la actividad de análisis de requerimientos se lleva a cabo principalmente al inicio y en la elaboración, mientras que la gestión de configuración y del proyecto se lleva a cabo a lo largo de todo el proyecto.

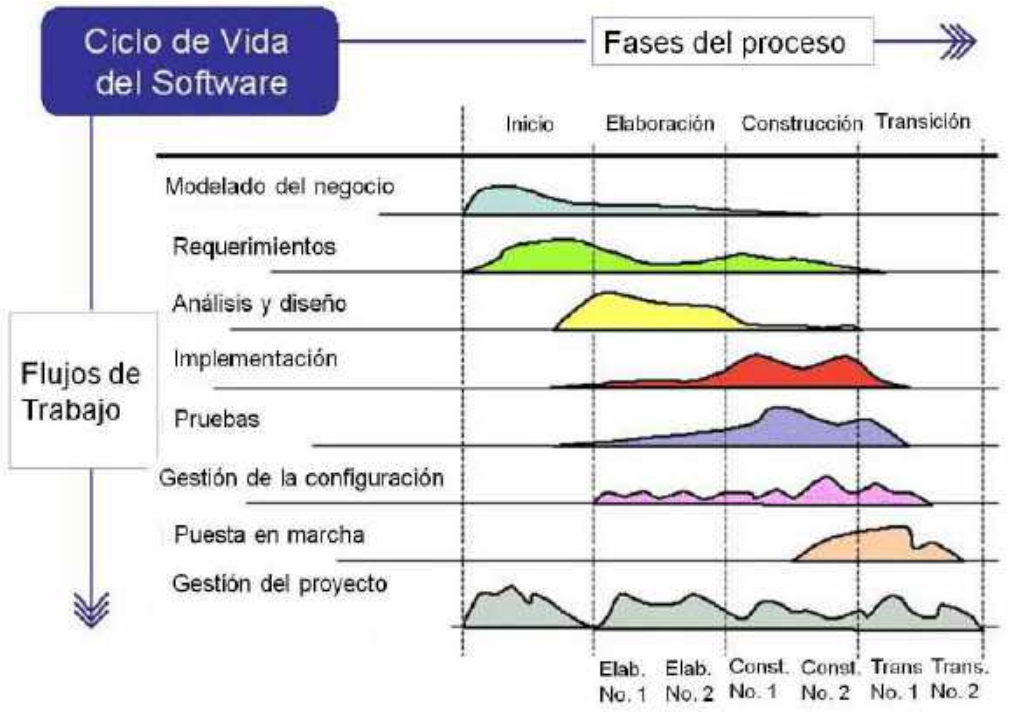


Figura 12-6: Combinación de las fases del Proceso Unificado con los flujos de trabajo[12]

En el Proceso Unificado se incluye la utilización del software en un entorno de usuario como parte del proceso, y se le nombra la *puesta en marcha*. Cada una de las fases se divide a su vez en varias iteraciones (la de inicio solo consta de varias iteraciones en proyectos grandes), como se ilustra en la Figura 12-7. Estas iteraciones ofrecen como resultado un *incremento* del producto desarrollado que añade o mejora las funcionalidades del sistema en desarrollo.

[12] https://i13.servimg.com/u/f13/11/74/78/81/desarr10.png

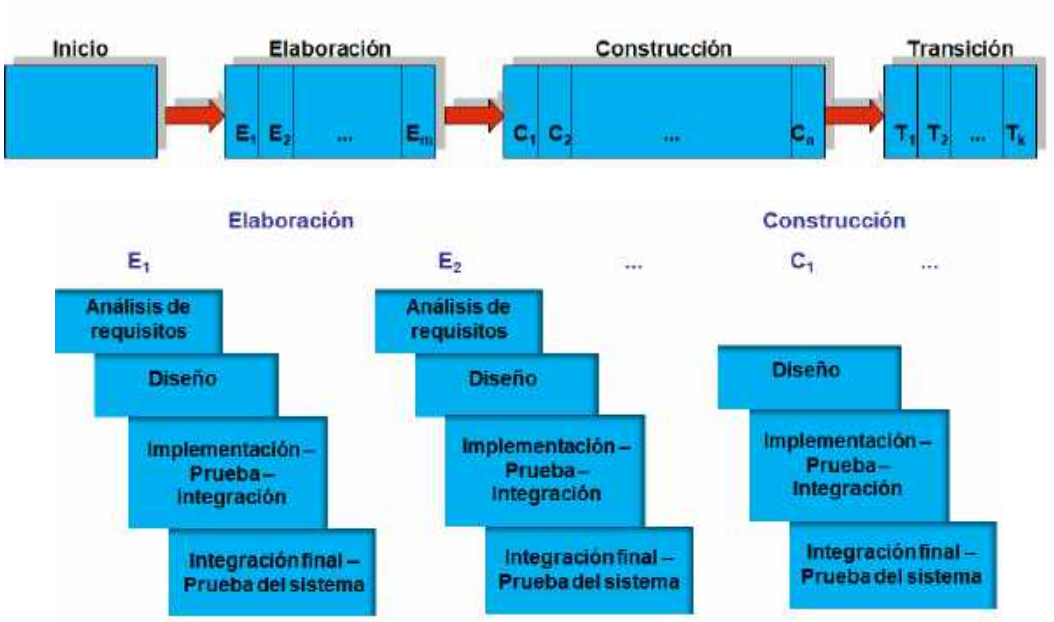


Figura 12-7: Las fases pueden ser iterativas

Los requerimientos se trabajan desde la fase de inicio, y muy especialmente durante la fase de elaboración, pues el objetivo es tener claro por lo menos un 80% del sistema que se requiere construir [Wiegers, 2013].

Al final de cada una de las etapas del Proceso Unificado se debe entregar un producto importante llamado ***hito*** (en inglés: *milestone*). Al final del *inicio*: se entregan los objetivos y definición del alcance del proyecto. Al final de la *elaboración*: se entrega la arquitectura del sistema. En cada iteración de la *construcción*: se entrega un producto con la función anterior más el incremento correspondiente a la nueva iteración, de tal forma que al final de la *construcción* se obtiene la versión inicial del sistema con capacidad operacional, es decir, con toda la funcionalidad requerida. Al final de la *transición*: se entrega el producto completamente funcional [Ambler, 2005].

Perspectiva práctica.- Desde esta perspectiva, se tiene un compendio de consejos a los que se les llama *buenas prácticas*, los cuales enlistamos a continuación:

1.- *Desarrollar el software de forma iterativa*.- Planificar incrementos del sistema basándose en las prioridades del usuario y desarrollo. Entregar las características del sistema que tengan la más alta prioridad al inicio del proceso de desarrollo.

2.- *Gestionar los requerimientos*.- Documentar explícitamente los requerimientos del cliente y mantenerse al tanto de los cambios en estos requerimientos. Analizar el impacto de los cambios en el sistema antes de aceptarlos.

3.- *Reutilizar el software*.- Estructurar la arquitectura del sistema en componentes reutilizables en la mayor medida posible

12.3.2 Ventajas y desventajas del Proceso Unificado

El Proceso Unificado es flexible, por lo que se puede adaptar a diferentes situaciones. Aunque fue concebido inicialmente para sistemas Orientados a Objetos, en algunos casos vale la pena aplicarlo a situaciones no Orientadas a Objetos y obtener algunas de sus ventajas como son [Ambler, 2005]:

- Hacer frente a los riesgos de cambios en los requerimientos.
- Disminuir el riesgo financiero al hacer entregas parciales de software funcional que puede probarse y ser evaluado por el cliente.

Sin embargo, el Proceso Unificado no es un proceso apropiado para todos los tipos de desarrollo, y requiere de alta capacitación de los participantes.

12.3.3 Resumen del Proceso Unificado

Las características del Proceso Unificado según [Ambler, 2005] son:

- Visto a lo largo de todo el proyecto, es *serial* en el tiempo: comienza con la etapa de inicio, luego la etapa de elaboración, después la etapa de construcción y al final la etapa de transición.
- Visto en cada etapa es *iterativo*: la etapa puede estar compuesta de varias entregas. Hay entregas parciales del producto, las funcionalidades se van incluyendo de manera incremental.
- Se apoya en buenas prácticas probadas en innumerables proyectos exitosos para una gran variedad de dominios.

Al final de cada una de las etapas del Proceso Unificado se debe entregar un producto importante (hito). Los hitos de cada fase se ilustran en la Figura 12-8.

Figura 12-8: Los hitos de las diferentes fases del Proceso Unificado

12.4 Resumen

La reusabilidad en el desarrollo de sistemas se basa en la existencia de componentes reutilizables, y tiene como finalidad usar de nuevo ideas, arquitecturas, diseños o código de una aplicación para construir otras. De aquí que dos aspectos clave a considerar en la calidad de un sistema de software a desarrollar, sean su reusabilidad y su extensibilidad.

El proceso de desarrollo de un sistema con reuso se enfoca en integrar los componentes reutilizables en el sistema en lugar de desarrollarlos desde cero. A la versión moderna de reuso se le llama *Desarrollo basado en componentes.*

El reuso del software tiene la ventaja obvia de reducir la cantidad de software a desarrollarse, por lo que se reducen los costos y los riesgos. Sin embargo, si se cambian mucho los requerimientos para adaptarse al sistema anterior, es posible que el nuevo sistema no cumpla las necesidades reales de los usuarios. Además, si las nuevas versiones de los componentes reutilizables no están bajo el control de la organización que los utiliza, se pierde el control sobre la evolución del sistema.

El Proceso Unificado (RUP) se concibió para el diseño orientado a objetos. Sin embargo, puede usarse para el paradigma estructurado. El RUP es un modelo híbrido porque reúne elementos de todos los modelos de procesos genéricos (cascada, evolutivo y reuso). Sus características predominantes son el desarrollo iterativo, el desarrollo dirigido por riesgos y el desarrollo incremental del sistema. El Proceso Unificado visualiza al sistema desde tres perspectivas: las fases (perspectiva dinámica), los flujos de trabajo (perspectiva estática) y las buenas prácticas a utilizar durante el desarrollo del sistema (perspectiva práctica).

12.5 Cuestionario

1.- ¿Cuáles son los elementos que se pueden reutilizar en el desarrollo de un nuevo sistema de software?

2.- Explica en qué consisten las dos ramas del Desarrollo Basado en Componentes

3.- ¿En qué consiste el *análisis de componentes*?

4.- ¿Cuál es el objetivo de las líneas de productos?

5.- ¿Cuál es la característica principal de los productos COTS?

6.- ¿Cuáles son las ventajas y desventajas de la reutilización del software?

7.- ¿Cuáles son las fases del Proceso Unificado a través del tiempo?

8.- ¿Qué son los flujos de trabajo del Proceso Unificado?

9.- ¿Qué es un ***hito***?

10.- Menciona las tres buenas prácticas del RUP

13 Tercer Caso de Estudio: Desarrollo del Sistema III con reutilización

13.1 Objetivos del capítulo

Objetivo general:

- Aplicar los conceptos estudiados en el capítulo anterior, en el desarrollo de un sistema con reutilización de componentes.

Objetivos específicos:

- Conocer, con un ejemplo, cómo se identifican los requerimientos similares en dos proyectos diferentes.
- Comprender, mediante un ejemplo práctico, la forma en la que se adaptan componentes de otros sistemas para reutilizarlos en uno nuevo.

13.2 Introducción

[13]En este capítulo reutilizaremos algunos de los elementos construidos para el Sistema II (*Control escolar de la Academia Patito*) en un sistema similar, al que llamaremos Sistema III. El Sistema III servirá para el *Control de citas de un consultorio dental.* Este sistema no requiere interfaz gráfica, es decir, funciona en la consola.

Para construir un sistema nuevo reutilizando un sistema anterior, no solo se reutiliza el código, sino también los diagramas, el formato de los requerimientos, el diseño etc.

A lo largo del desarrollo del sistema III mencionaremos los elementos que se reutilizaron.

A continuación, se presentan los requerimientos del sistema completo.

13.2.1 Visión global de los Requerimientos del Sistema III

Con el ***Sistema III*** se pueden hacer las siguientes operaciones:

- Alta y consulta de pacientes
- Alta, consulta y cancelación de una cita

El sistema terminará de ejecutarse cuando el usuario seleccione la opción "salir" del menú principal.

Figura 13-1: Diagrama de casos de uso del Sistema III

[13] http://www.educando.edu.do/portal/wp-content/uploads/2016/01/Jobs-MATERIALS-page-005.jpg

En la Figura 13-1 se ilustra el diagrama de casos de uso del Sistema III. Basándonos en este diagrama describimos los menús del sistema. Primero se presentará un *menú principal* con las siguientes opciones:

Selecciona la opción deseada:
1.- Pacientes
2.- Citas
3.- Salir

Habrá un *menú de pacientes* que desplegará lo siguiente:

Selecciona la opción deseada:
1.- Dar de alta a un paciente
2.- Consultar pacientes
3.- Regresar

También habrá un *menú de citas* que desplegará lo siguiente:

Selecciona la opción deseada:

1.- Dar de alta un mes de trabajo
2.- Consultar las citas de un mes
3.- Agendar una cita
4.- Cancelar una cita
5.- Regresar

En la Figura 13-2 se ilustra el Diagrama de Transición entre Interfaces de Usuario (DTIU) del Sistema III.

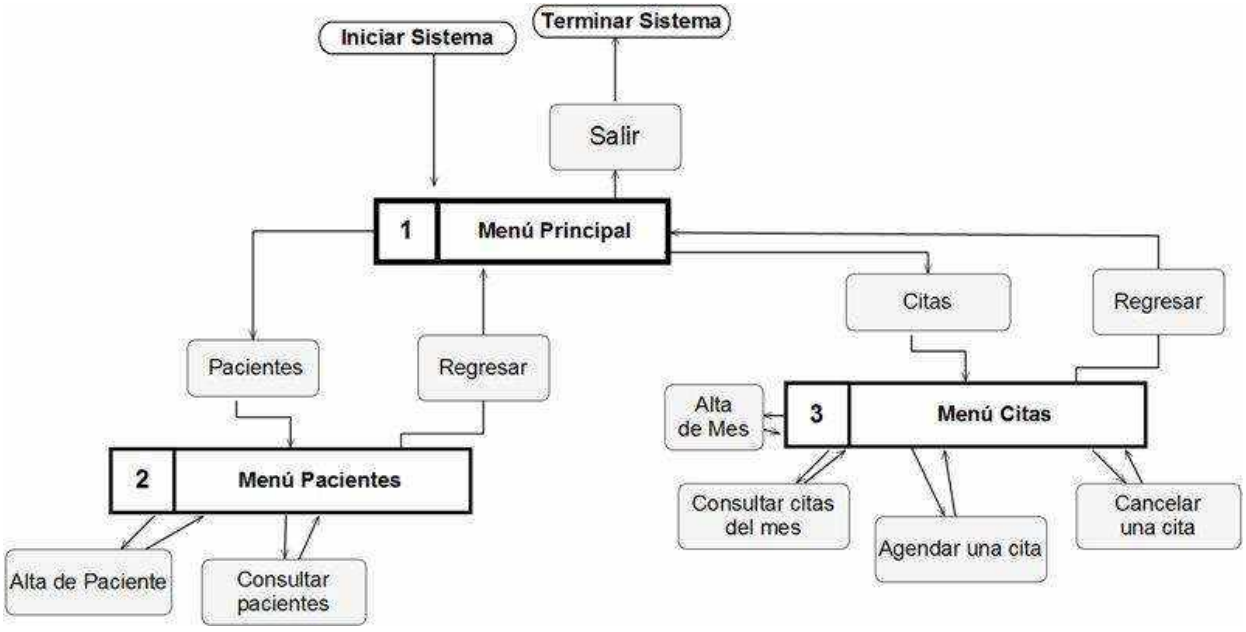


Figura 13-2: Diagrama de Transición entre Interfaces de Usuario (DTIU) del Sistema III

Reuso: Nótese que para elaborar el DTIU del Sistema III, nos basamos en el DTIU del Sistema II y se hicieron las modificaciones necesarias. De esta manera no hubo que comenzar a dibujar todo el diagrama desde cero.

13.3 Requerimientos del Sistema III: Control de citas de un consultorio dental

R1.- Requerimientos de funcionamiento del *menú Pacientes*

R1.1.- Al seleccionar "Alta de paciente" se solicita al usuario el nombre completo del paciente, y su teléfono. Estos datos se guardan en un archivo llamado "Pacientes.txt".

R1.2.- Se notifica al usuario cuando el paciente se haya registrado con éxito en el archivo.

R1.3.- Al seleccionar "Consultar Pacientes" se desplegarán:

- El nombre
- El teléfono

De todos los pacientes registrados.

R2.- Requerimientos de funcionamiento del *menú Citas*

R2.1.- Funcionamiento de "Alta de un mes de trabajo"

R2.1.1.- Al seleccionar "Alta de un mes de trabajo" se solicita al usuario el año y el nombre del mes.

R2.1.2.- Se generará un archivo cuyo nombre estará compuesto por: "añoMesCitas.txt" en el que se agendarán las citas del mes.

R2.1.3.- Se notificará al usuario si el mes de trabajo se dio de alta con éxito o no.

R2.2.- Funcionamiento de "Agendar una cita"

R2.2.1.- Al seleccionar "Agendar una cita" se solicita el teléfono del paciente, el año y el mes en el que se desea agendar la cita, se busca el archivo "añoMesCitas.txt" y se agrega en este archivo:

- El día
- La hora
- El teléfono del cliente

R2.2.2.- En caso de que no se encuentre el archivo "añoMesCitas.txt" se le indicará al usuario.

R2.2.3.- En caso de que no se encuentre registrado el teléfono en "Pacientes.txt" se indicará al usuario.

R2.2.4.- En caso de que el horario ya esté ocupado se indicará al usuario.

R2.3.- Funcionamiento de "Cancelar una cita"

R2.3.1.- Al seleccionar “Cancelar una cita” se solicita el teléfono del paciente, el año y el mes en el que está agendada la cita, se busca el archivo “añoMesCitas.txt” y se borra el día, la hora y el teléfono del paciente en ese archivo.

R2.3.2.-En caso de que no se encuentre el archivo “añoMesCitas.txt” se le indicará al usuario.

R2.3.3.-En caso de que no se encuentre el teléfono proporcionado en el mes de trabajo se le indicará al usuario.

13.4 Desarrollo por etapas del Sistema III

Desarrollaremos el **Sistema III** con un modelo incremental en 2 etapas. Comenzaremos con la etapa 1 como se ilustra en la Figura 13-3.

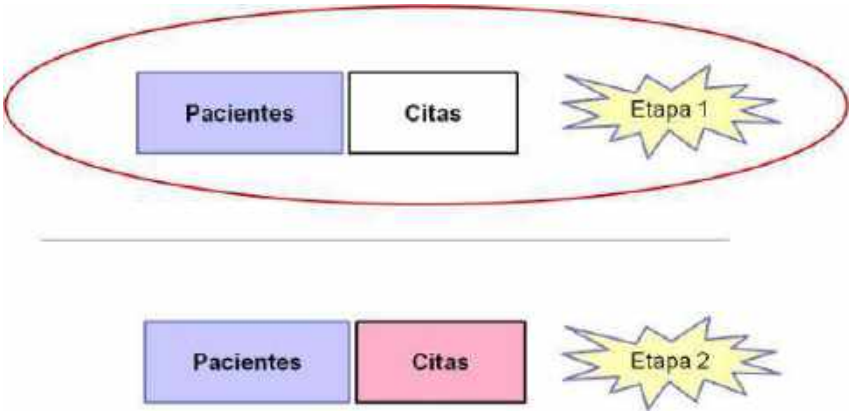


Figura 13-3: Primera de las 2 etapas del desarrollo incremental del Sistema III

13.4.1 Diseño con reutilización para la etapa 1: *menú de pacientes*

13.4.1.1 *Primer bloque de la etapa 1*

En este primer bloque de esta primera etapa codificaremos el esqueleto de los menús, y en el segundo bloque la funcionalidad *Alta de paciente* y *Consultar pacientes* que son las opciones del "Menú Pacientes", como se ilustra en el DTIU de la Figura 13-4.

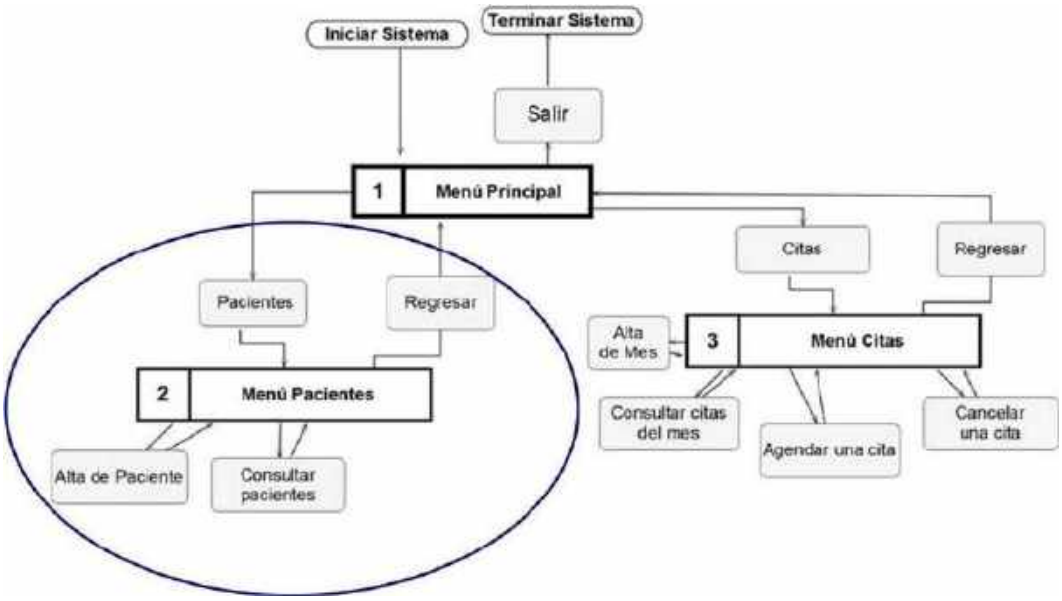


Figura 13-4: Diagrama de Transición Entre Interfaces de Usuario (DTIU) Etapa 1

Diseño y codificación

Reutilizaremos la clase abstracta `Menu` del Sistema II, pero ahora los menús serán los que se ilustran en la Figura 13-5. La codificación será muy similar a la del esqueleto del Sistema II.

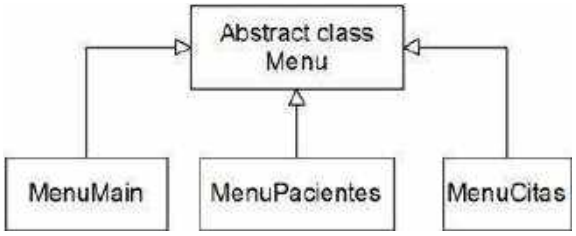


Figura 13-5: Diagrama de Clases de los menús del Sistema III

Clase `MenuMain`:

```
package sistemadentista;
import utilerias.Menu;
public class MenuMain extends Menu{
    MenuMain(){
          opciones = new String[3];
          opciones[0] = "Pacientes";
          opciones[1] = "Citas";
          opciones[2] = "Salir";
       }
}
```

Clase `MenuPacientes`:

```
package sistemaDentista;
import utilerias.Menu;
public class MenuPacientes extends Menu{
```

```
    MenuPacientes(){
          opciones = new String[3];
          opciones[0] = "Dar de alta a un paciente";
          opciones[1] = "Consultar pacientes";
          opciones[2] = "Regresar";
       }
}
```

Clase `MenuCitas`:

```
package sistemaDentista;
import utilerias.Menu;
public class MenuCitas extends Menu{
    MenuCitas(){
        opciones = new String[5];
             opciones[0] = "Dar de alta a un mes de trabajo";
             opciones[1] = "Consultar las citas de un mes";
             opciones[2] = "Agendar una cita";
             opciones[3] = "Cancelar una cita";
             opciones[4] = "Regresar";
    }
}
```

Al igual que en el Sistema II, primero codificamos el esqueleto sin funcionalidad.

```
package sistemadentista;
public class SistemaDentista {
    /*
     * Sistema que administra las citas
     * en un consultorio
     */
    public static void main(String[] args) {
        MenuMain  menuMain = new MenuMain();
        File fPacientes = new File( "Pacientes.txt" );

        while( true ){
            int opcion = menuMain.opcion();

            switch( opcion ){
                case 1: pacientes( fPacientes );
                        break;
                case 2: citas(  );
                        break;
                case 3: return;
             }
         }
     }
private static void pacientes() {
        MenuPacientes  menuPacientes = new MenuPacientes();
        while( true ){
            int opcion = menuPacientes.opcion();
            switch( opcion ){
                case 1: altaPaciente();
                        break;
                case 2: consultarPacientes();
                        break;
                case 3: return;
            }
        }
    }
    private static void altaPaciente() {
```

Falta codificar la funcionalidad

```
        System.out.println("El paciente quedó registrado ");
    }
    private static void consultarPacientes() {
        System.out.println(" Los pacientes registrados son: ");
    }
```

Antes de codificar el segundo bloque se debe probar que los menús codificados hasta ahora funcionen correctamente.

13.4.1.2 Segundo bloque de la etapa 1

Diseño y codificación

Para codificar la funcionalidad *Alta de Paciente* nos basaremos en el diagrama de flujo de la Figura 13-6.

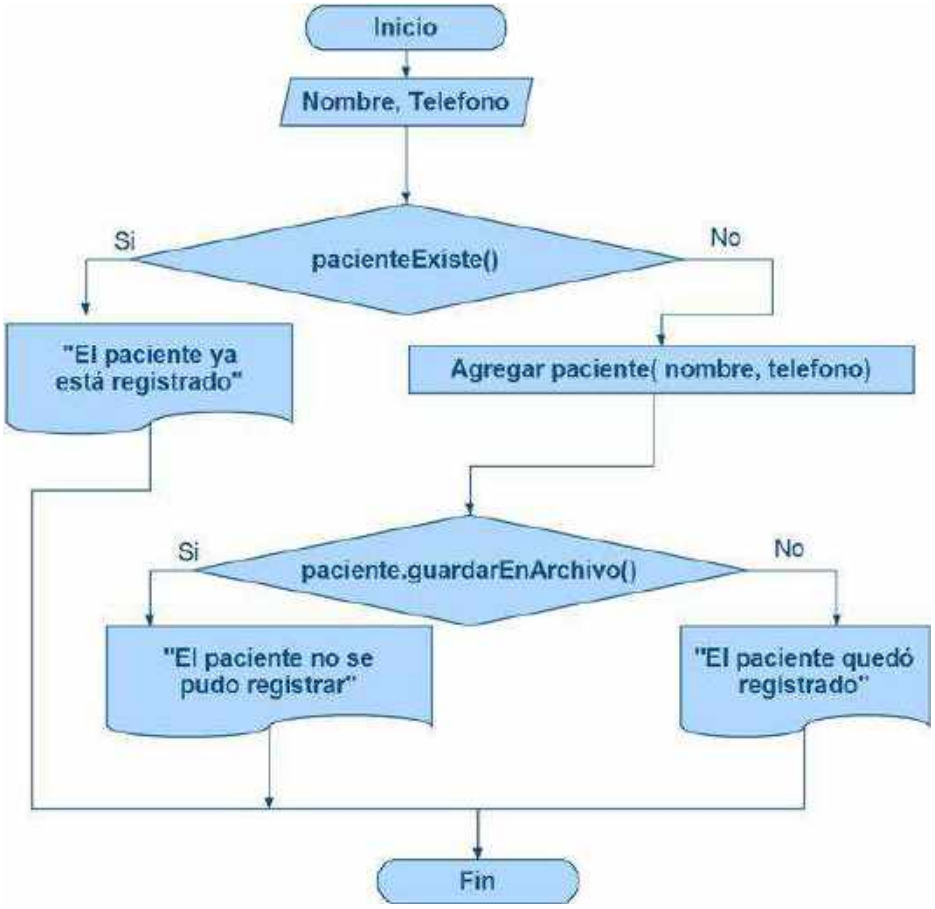


Figura 13-6: Diagrama de flujo de "Alta de un paciente"

Los métodos y atributos de la clase `Paciente` se ilustran en la Figura 13-7.

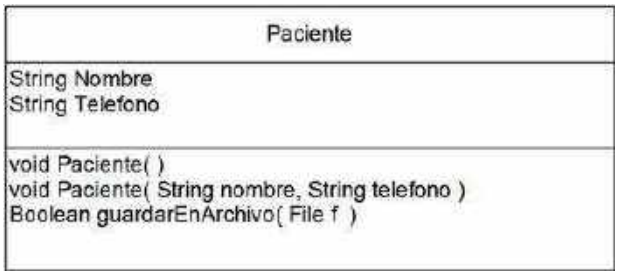


Figura 13-7: La clase `Paciente`

A continuación presentamos el código de la clase `Paciente` (no se muestran los setters ni los getters).

```
public class Paciente {
  private String nombre;
  private String telefono;

  public Paciente(){

  }

  public Paciente(String nombre, String telefono) {
    this.nombre = nombre;
    this.telefono = telefono;
  }

  public Boolean guardarEnArchivo( File f ){
    Boolean exito = false;
    try{
      // append = true para agregar sin sobreescribir
      FileWriter w = new FileWriter( f, true );
      BufferedWriter bw = new BufferedWriter(w);
      PrintWriter salida = new PrintWriter( bw );
      // Escribimos los datos en archivo
      // Separando los campos por comas
      salida.println( nombre + "," + telefono );
      salida.close();
      bw.close();
      exito = true;
    }catch(IOException e){};
      return exito;
 }
```

Es importante no poner espacio después de la coma (porque luego vamos a comparar cadenas de caracteres)

Podemos reutilizar el código de

```
private static void altaAlumno( File f ) { . . .}
private static void consultarAlumnos( File f ) { . . .}
private static boolean alumnoExiste( int matr ) { {
```

para codificar

```
private static void altaPaciente( File f ) { . . .}
private static void consultarPacientes( File f ) { . . .}
private static boolean pacienteExiste( String tel ) {
```

Solo tendremos que hacer algunas modificaciones. Dejamos al lector la codificación de los métodos `altaPaciente( )`, `consultarPacientes()` y `pacienteExiste( )`.

En la Figura 13-8 se ilustra que tanto *Alta de un paciente* como *Consultar pacientes* funciona correctamente.

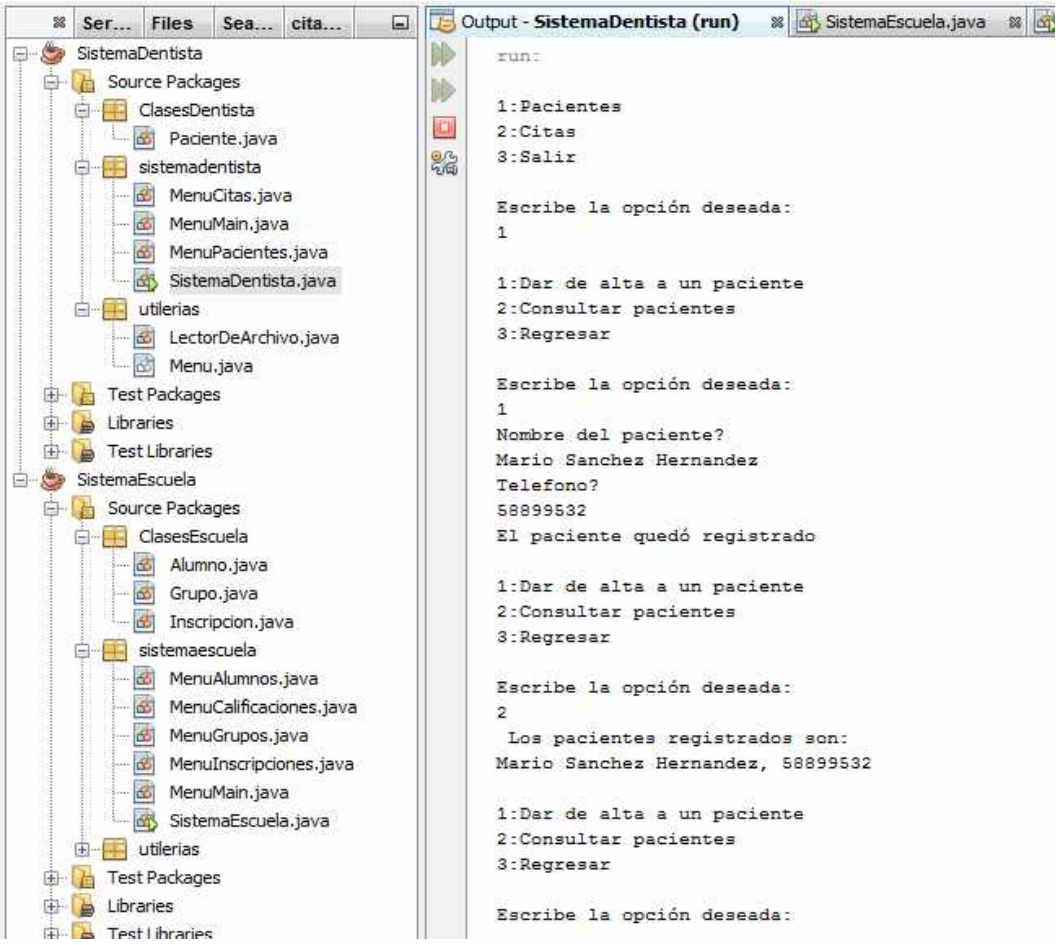


Figura 13-8: Funcionalidad "Alta de un paciente" y "Consultar pacientes"

Pruebas de validación

Pruebas del menú pacientes, opción: Dar de alta un paciente

I.1.- Se selecciona la opción 1 del menú principal. Luego la opción 1 del menú de pacientes. Se proporciona la el nombre y el teléfono de un paciente.

El sistema deberá dar de alta el paciente en el archivo pacientes.txt

I.2.- Se selecciona la opción 1 del menú principal. Luego la opción 1 del menú de pacientes. Se proporciona la el nombre y el teléfono de un paciente que ya está dado de alta.

El sistema deberá decir que el paciente ya está dado de alta.

Pruebas del menú pacientes, opción: Consultar pacientes

I.3.- Se selecciona la opción 1 del menú principal. Luego la opción 2 del menú de pacientes.

El sistema deberá mostrar el nombre y teléfono de los pacientes dados de alta en el archivo pacientes.txt

13.4.2 Diseño con reutilización para la etapa 2: *menú de citas*

Ahora desarrollaremos la etapa 2 del modelo incremental, como se ilustra en la Figura 13-9.

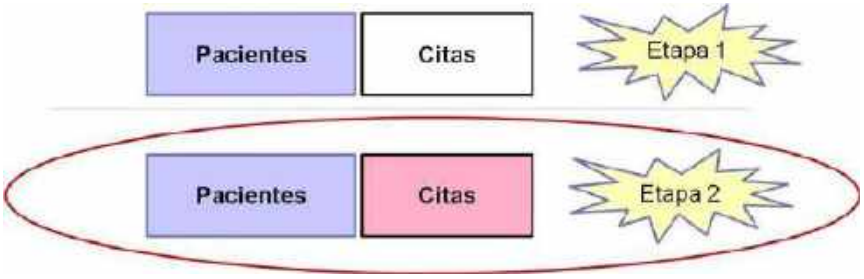


Figura 13-9: Segunda de las 2 etapas del desarrollo incremental del Sistema III

En la segunda etapa desarrollaremos todas las funcionalidades del menú de citas, que se pueden observar en la Figura 13-10. Como son cuatro, dividiremos la segunda etapa en cuatro bloques, uno para cada funcionalidad.

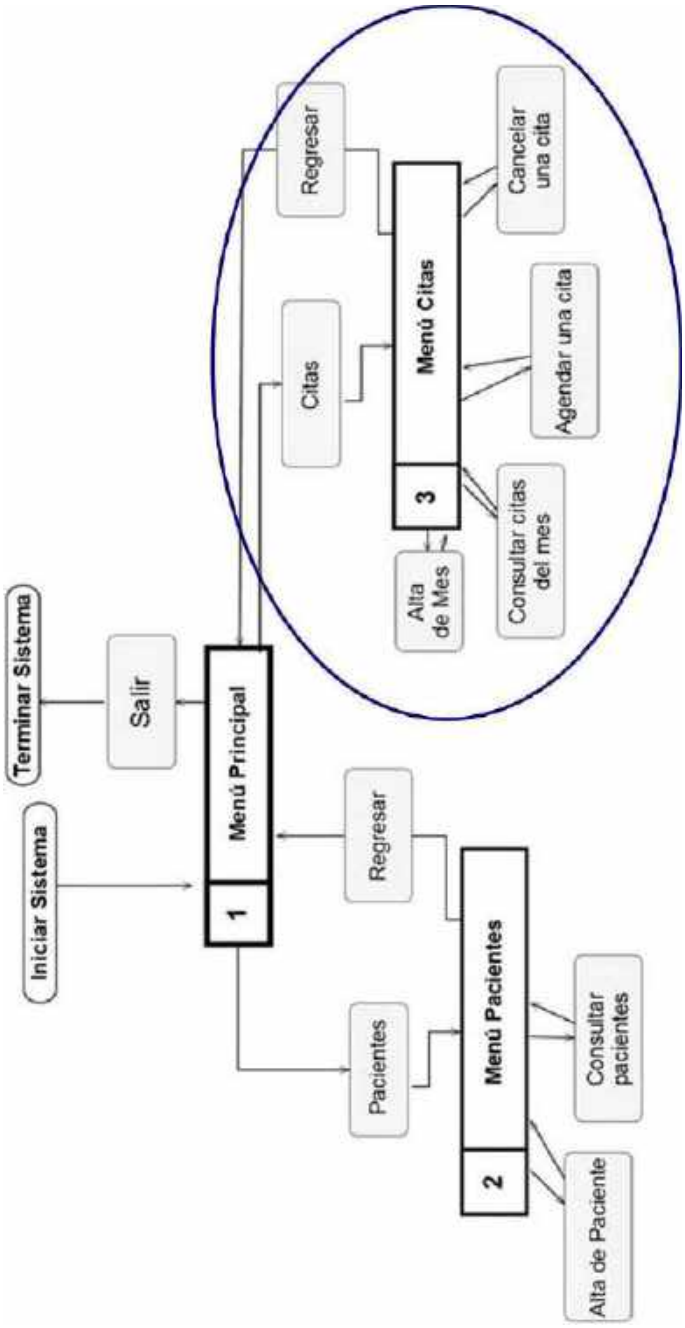


Figura 13-10: Diagrama de Transición Entre Interfaces de Usuario (DTIU) Etapa 2

13.4.2.1 Primer bloque de la etapa 2

En este primer bloque de la segunda etapa codificaremos la funcionalidad Alta *de un mes de trabajo.*

Diseño

Para codificar la funcionalidad *Alta de un mes de trabajo* nos basaremos en el diagrama de flujo de la Figura 13-11.

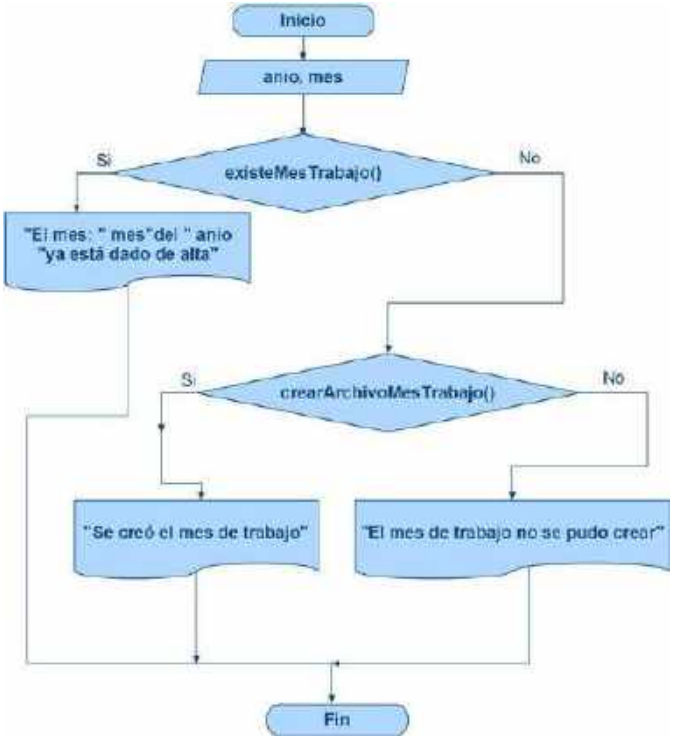


Figura 13-11: Diagrama de flujo de "Alta de un mes de trabajo"

Reuso: Para elaborar este diagrama reutilizamos el diagrama de "Alta de un Grupo" del Sistema II, el cual es muy similar al funcionamiento del alta de un mes de trabajo.

Codificación

Podemos reutilizar el código de

```
private static void altaGrupo( File f ) { . . .}
```

para codificar

```
private static void altaMesTrabajo( File f ) { . . .}
```

A continuación presentamos el código de `altaMesTrabajo()`

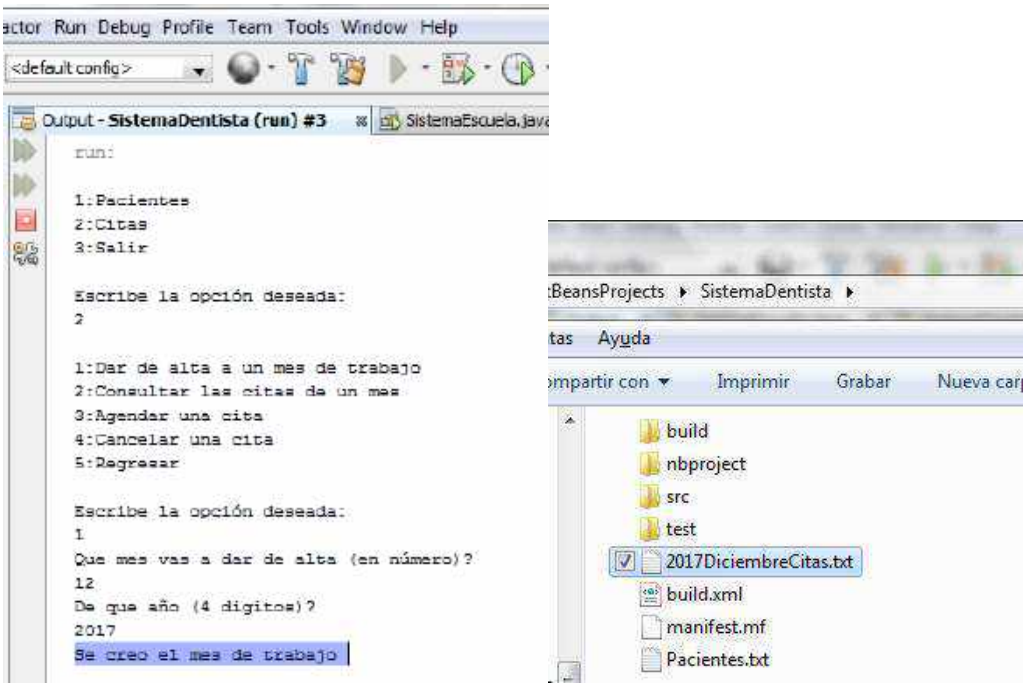
```
private static void altaMesTrabajo() {
    int anio;
    int mes;
    String mesString;
    Scanner input = new Scanner( System.in );
```

```
    do{
       System.out.println( "Que mes vas a dar de alta (en número)? ");
       mes = input.nextInt();
    }while( mes < 1 || mes > 12 );

    mesString = mesLetra( mes );

    System.out.println( "De que año (4 digitos)? ");
    anio = input.nextInt();

    if( existeMesTrabajo( nombreArchivo )){
        System.out.println( mesString + " de " + anio
                            + " ya esta dado de alta" );
        return;
    }
    // El nombre del archivo del mes de trabajo
    // es la concatenación del año con el mes con "Citas.txt"
    String nombreArchivo = anio + mesString + "Citas.txt";

    File fMesDeTrabajo  = new File( nombreArchivo );
    if ( creaArchivoMesTrabajo( fMesDeTrabajo ) )
         System.out.println( "Se creó el mes de trabajo " );
    else
         System.out.println( "El mes de trabajo no se pudo crear" );
  }

  private static String mesLetra( int mes ){
     switch( mes ){
        case  1: return "Enero";
        case  2: return "Febrero";
        case  3: return "Marzo";
        case  4: return "Abril";
        case  5: return "Mayo";
        case  6: return "Junio";
        case  7: return "Julio";
        case  8: return "Agosto";
        case  9: return "Septiembre";
        case 10: return "Octubre";
        case 11: return "Noviembre";
        case 12: return "Diciembre";
      }
     return null;
  }
}
```

El mes se pide en número y se convierte a letras para evitar inconsistencias en los nombres de archivo.

Para codificar `creaArchivoMesTrabajo( )` reutilizamos `creaArchivoAlumnos( File f )` , tal como está en el Sistema II, solo es necesario cambiar el nombre del método.

En la Figura 13-12 a) se observa el programa corriendo con la funcionalidad *Alta de un mes de trabajo.* Para verificar que ésta funciona correctamente, es necesario comprobar que se creó el archivo correspondiente en la carpeta del proyecto, como se ilustra en la Figura 13-12 b).



a) Programa corriendo ***b) Archivo del mes de trabajo***

Figura 13-12: La funcionalidad Alta de un mes de trabajo, el archivo se genera correctamente

Pruebas de validación

Pruebas del menú citas opción: Dar de alta un mes de trabajo

I.1.- Se selecciona la opción 2 del menú principal. Luego la opción 1 del menú de citas. Se proporciona el mes y el año del mes de trabajo que se desea dar de alta

El sistema deberá generar el archivo año*Mes*Citas.txt

I.2.- Se selecciona la opción 2 del menú principal. Luego la opción 1 del menú de citas. Se proporciona el mes y el año de un mes de trabajo que ya está dado de alta

El sistema deberá decir que el mes de trabajo ya está dado de alta.

13.4.2.2 Segundo bloque de la etapa 2

En el segundo bloque de esta segunda etapa codificaremos la funcionalidad Agendar *una cita.*

Diseño y codificación

La clase que se necesita para registrar una cita en el archivo de un mes de trabajo es: `Cita`. En la Figura 13-13 se describen sus atributos y sus métodos.

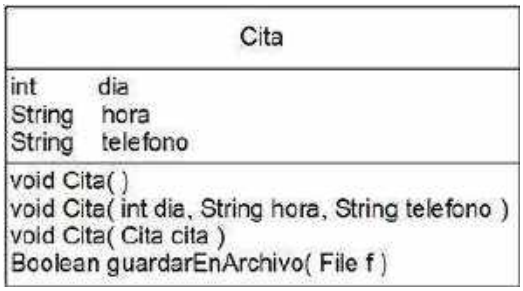


Figura 13-13: La clase `Cita`

En la clase `Cita` reutilizamos el método `guardarEnArchivo( )` que usamos en las clases `Alumno`, `Grupo` e `Inscripcion` del Sistema II. Al igual que en la clase `Paciente`, es importante no poner espacio después de la coma al momento de guardar los datos en el método `guardarEnArchivo ()` , porque luego vamos a comparar cadenas de caracteres y el espacio en blanco hace que dos palabras iguales sean diferentes. Por ejemplo, "cadena" no es igual a " cadena" :

```
salida.println( dia + "," + hora + "," + telefono );
```

Para codificar la funcionalidad *Agendar una cita* nos basaremos en el diagrama de flujo de la Figura 13-14.

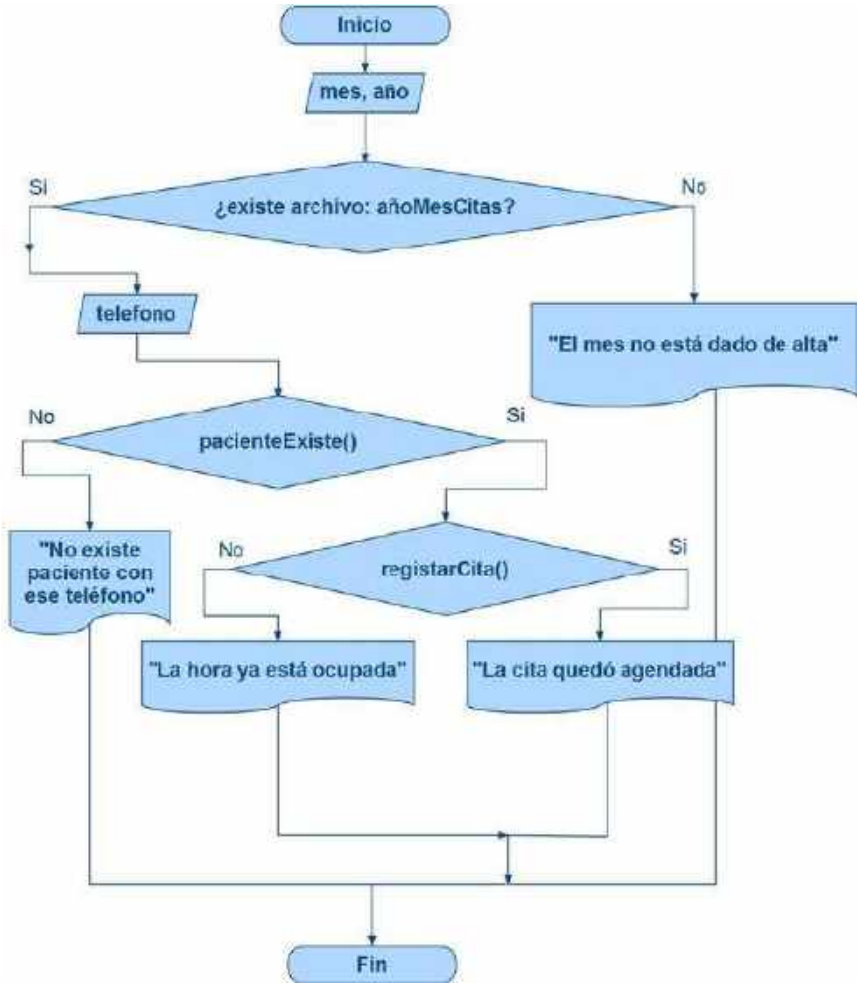


Figura 13-14: Diagrama de flujo de "Agendar una cita"

Reuso: Para elaborar este diagrama reutilizamos el diagrama de "Alta de alumno en un grupo" del Sistema II, el cuál es muy similar a la funcionamiento de "Agendar una cita".

A continuación el código de `agendarCita( )`

Nótese que en el siguiente código se pregunta primero por las condiciones de excepción, así evitamos varios `if-else` anidados (ver la regla de codificación de la sección 6.2.5.2)

```
private static void agendarCita() {
    int     anio;
    int     mes;
    String  mesString;
    String  telefono;
    Scanner input = new Scanner( System.in );

    do{
       System.out.println( "En qué mes es la cita (en número)? ");
       mes = input.nextInt();
    }while( mes < 1 || mes > 12 );
```

```
    mesString = mesLetra( mes );

    System.out.println( "De que año (4 digitos)? ");
    anio = input.nextInt();
    // se agrega la siguiente instrucción para borrar el enter
    // que quedó guardado en el buffer (problema de la clase Scanner)
    String enter = input.nextLine();

    // El nombre del archivo del mes de trabajo
    // es la concatenación del año con el mes con "Citas.txt"
    String nombreArchivo = anio + mesString + "Citas.txt";

    if( !existeMesTrabajo( nombreArchivo )){
        System.out.println( "El mes no está dado de alta" );
        return;
    }

    System.out.println( "Teléfono del paciente? ");
    telefono = input.nextLine();
    // Código para eliminar posibles espacios en blanco
    while( telefono.startsWith( " " ) )
      telefono = telefono.substring( 1 );

    if( !pacienteExiste( telefono )){
        System.out.println("No existe paciente con ese teléfono");
        return;
    }

    if( registrarCita( nombreArchivo, telefono ))
        System.out.println("La cita quedó agendada");
    else
        System.out.println("La hora ya está ocupada");
  }
```

El código del método `agregaAlumnoAlGrupo( )` del Sistema II nos puede servir de base para codificar `registrarCita( )`, sin embargo, hay que hacer un rediseño, porque el funcionamiento de estos dos métodos tiene diferencias substanciales. En la Figura 13-15 se ilustra el diagrama de flujo del algoritmo para registrar una cita.

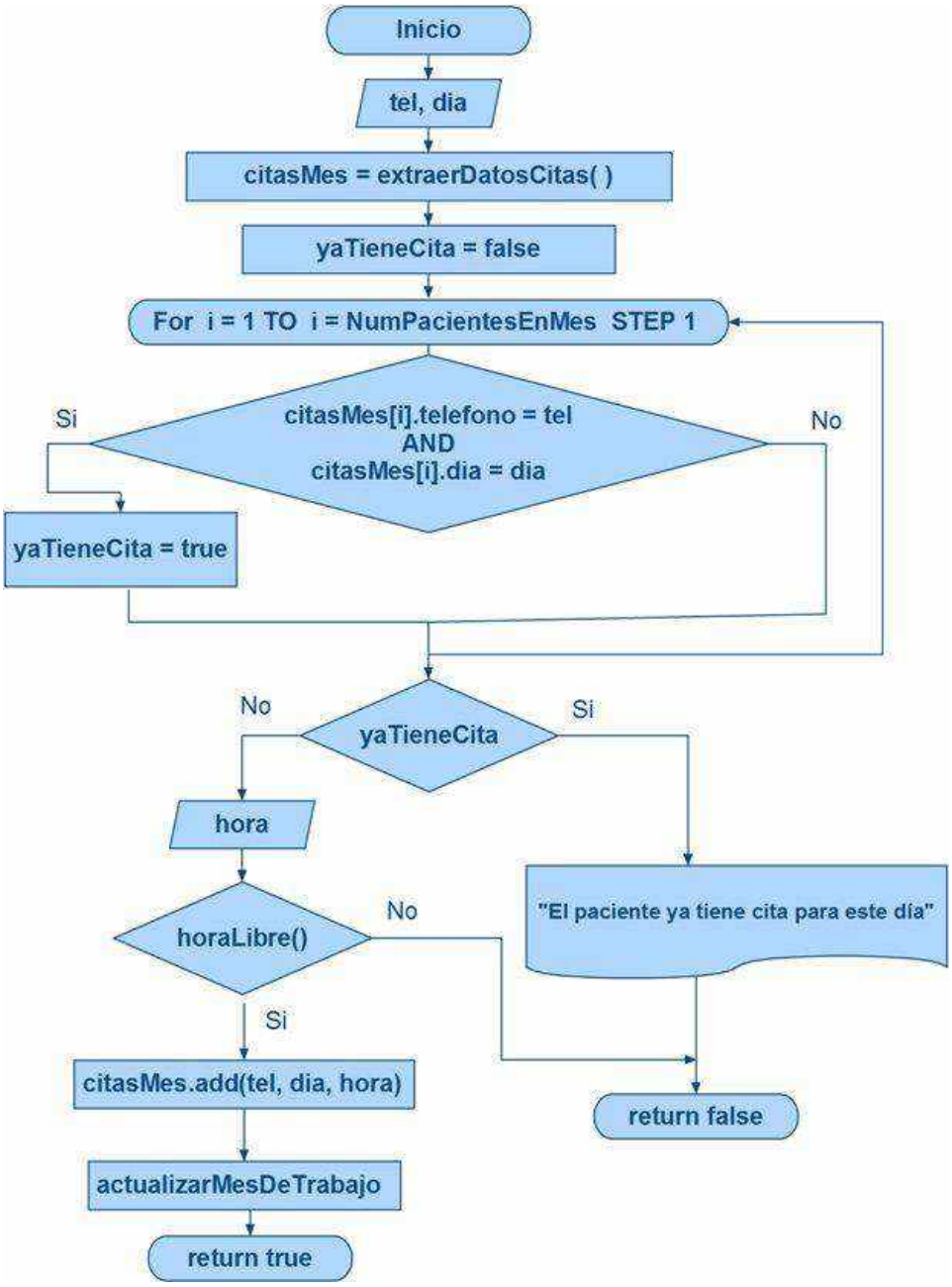


Figura 13-15: Diagrama de flujo de "Registrar una cita"

A continuación el código de `registrarCita( )`

```
private static boolean registrarCita( String nombreArchivo, String tel )
{
    int     dia;
    String     hora;
    Scanner input = new Scanner( System.in );

    System.out.println( "Qué dia quieres la cita (en número)? ");
    dia = input.nextInt();
    // se agrega la siguiente instrucción para borrar el enter
    // que quedó guardado en el buffer (problema de la clase Scanner)
```

```
    String enter = input.nextLine();

    File fCitasMes  = new File( nombreArchivo );
    Boolean exito = false;
    Boolean yaTieneCita = false;

    ArrayList<String> citasEnMesStr = new ArrayList<String>();
    ArrayList<Cita> citasEnMes = new ArrayList<Cita>();
    LectorDeArchivo lectorDeArchivo =
                                    new LectorDeArchivo( nombreArchivo );
    citasEnMesStr = lectorDeArchivo.LeerArchivo();
    citasEnMes = extraerDatosCitasEnMes( citasEnMesStr );

    for( int i=0; i < citasEnMes.size(); i++ ){
      if( citasEnMes.get(i).getTelefono().equals( tel )
          &&
          dia == citasEnMes.get(i).getDia() )
          yaTieneCita = true;
    }

    if( yaTieneCita ){
        System.out.println( "El paciente ya tiene cita para este día" );
        return false;
    }

    System.out.println( "A qué hora quieres la cita (hora:minutos)? ");
    hora = input.nextLine();

    if( horaLibre( dia, hora, citasEnMes ) ){
        Cita cita = new Cita( dia, hora, tel );
        exito = cita.guardarEnArchivo( fCitasMes );
        return exito;
    }
    else
        return false;

  }
```

Cuando comparamos Strings no usamos **==**, en su lugar usamos el método **`equals( )`**

Podemos reutilizar el código de

```
private static Boolean existeGrupoAlumnos( File f ) {
```

para codificar

```
private static Boolean existeMesTrabajo( File f ) {
```

Solo tendremos que hacer algunas modificaciones. Se deja al lector como ejercicio realizar estas modificaciones. Además habrá que codificar el método `horaLibre()`.

Para verificar que *Agendar una cita* funciona correctamente será necesario comprobar que la cita se agregó dentro del archivo correspondiente, como se ilustra en la Figura 13-16.

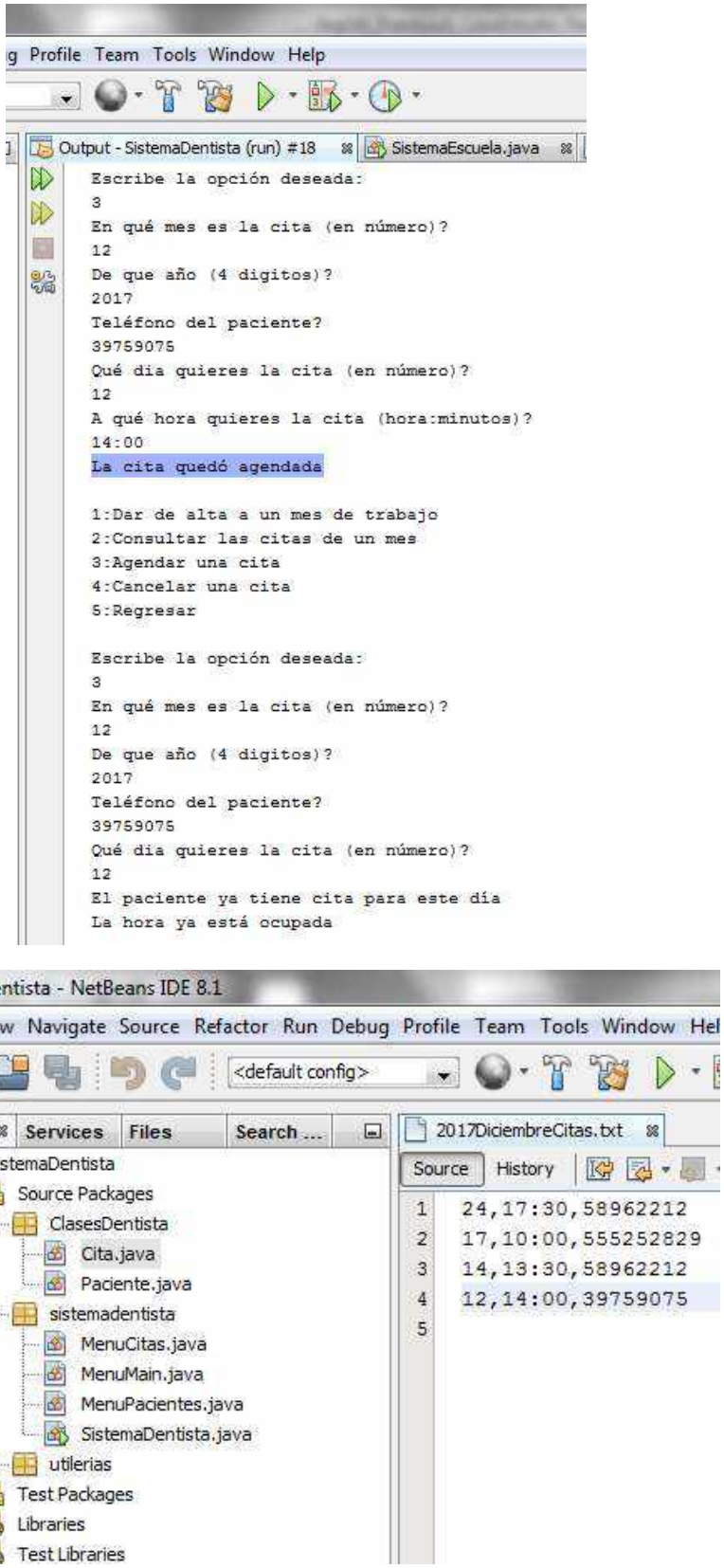


Figura 13-16: Funcionamiento correcto de "Agendar una cita"

Pruebas de validación

Pruebas del menú citas opción: Agendar una cita

I.1.- Se selecciona la opción 2 del menú principal. Luego la opción 3 del menú de citas. Se proporciona el mes y el año del mes en el que se desea hacer la cita. El mes de trabajo ya debe estar dado de alta previamente.

Posteriormente se da el teléfono de un paciente ya registrado.

A continuación, se proporciona el día y la hora de la cita.

El sistema deberá registrar los datos en el archivo año*Mes*Citas.txt y decir que **la cita quedó agendada**.

I.2.- Se selecciona la opción 2 del menú principal. Luego la opción 3 del menú de citas. Se proporciona el mes y el año del mes en el que se desea hacer la cita. El mes de trabajo ya debe estar dado de alta previamente. Posteriormente se da el teléfono de un paciente que no está registrado.

El sistema deberá decir que **no existe paciente con ese teléfono**.

I.3.- Se selecciona la opción 2 del menú principal. Luego la opción 3 del menú de citas. Se proporciona el mes y el año del mes en el que se desea hacer la cita. El mes de trabajo ya debe estar dado de alta previamente. Posteriormente se da el teléfono de un paciente que ya está registrado.

A continuación, se proporciona el día y la hora de la cita.

La hora de la cita ya deberá estar registrada en el archivo año*Mes*Citas.txt

El sistema deberá decir que **la hora ya está ocupada**.

I.4.- Se selecciona la opción 2 del menú principal. Luego la opción 3 del menú de citas. Se proporciona el mes y el año del mes en el que se desea hacer la cita. El mes de trabajo ya debe estar dado de alta previamente. Posteriormente se da el teléfono de un paciente que ya está registrado.

A continuación, se proporciona el día y la hora de la cita.

El día de la cita ya deberá estar registrado en el archivo año*Mes*Citas.txt , asociado con el teléfono proporcionado.

El sistema deberá decir que **el paciente ya tiene cita ese día**. (También se despliega que la hora ya está ocupada).

I.5.- Se selecciona la opción 2 del menú principal. Luego la opción 3 del menú de citas. Se proporciona el mes y el año del mes en el que se desea hacer la cita. El mes de trabajo NO está dado de alta previamente.

El sistema deberá decir que **el mes no está dado de alta.**

13.4.2.3 Tercer bloque de la etapa 2

En el tercer bloque de esta segunda etapa codificaremos la funcionalidad Consultar *citas del mes.*

Diseño y codificación

Podemos reutilizar el código de

```
private static void consultaGrupos( ) { . . .}
```

para codificar

```
private static void consultaMes( ) { . . .}
```

Solo tendremos que hacer algunas modificaciones. A continuación se presenta el código del método `ConsultarCitasMes()`.

```
private static void consultarCitasMes() {
    int     anio;
    int     mes;
    String  mesString;
    Scanner input = new Scanner( System.in );

    do{
       System.out.println( "De qué mes quieres consultar las
                            citas (en número)? ");
       mes = input.nextInt();
    }while( mes < 1 || mes > 12 );

    mesString = mesLetra( mes );

    System.out.println( "De que año (4 digitos)? ");
    anio = input.nextInt();
    String enter = input.nextLine();
    String nombreArchivo = anio + mesString + "Citas.txt";

    if( !existeMesTrabajo( nombreArchivo )){
        System.out.println( "El mes no está dado de alta" );
        return;
    }

    ArrayList<String> citas = new ArrayList<String>();
    LectorDeArchivo lectorDeArchivo =
                                    new LectorDeArchivo( nombreArchivo );
    citas = lectorDeArchivo.LeerArchivo();

    System.out.println(" Las citas agendadas en: "+ mesString + " de "
                       + anio + " son las siguientes: ");
    for( int i=0; i< citas.size() ; i++){
         System.out.println(citas.get(i));
    }
  }
```

En la Figura 13-17 se ilustra que *Consultar citas del mes* funciona correctamente.

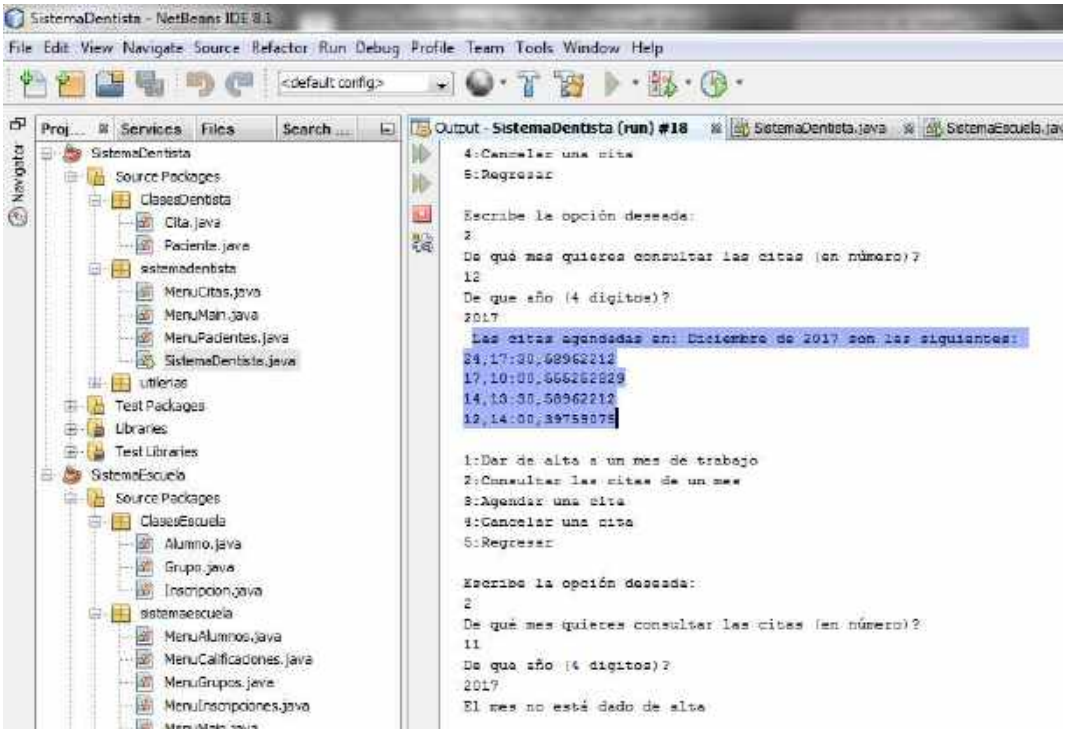


Figura 13-17: Funcionamiento correcto de la funcionalidad "Consultar citas del mes"

Pruebas de validación

Pruebas del menú citas opción: Consultar citas

I.1.- Se selecciona la opción 2 del menú principal. Luego la opción 2 del menú de citas. Se proporciona el mes y el año del mes del que se desea consultar las citas. El mes de trabajo ya debe estar dado de alta previamente.

El sistema deberá mostrar el día, la hora y el teléfono de cada una de las citas registradas.

I.2.- Se selecciona la opción 2 del menú principal. Luego la opción 2 del menú de citas. Se proporciona un mes o un año del mes que no tenga un mes de trabajo asociado.

El sistema deberá decir que **el mes no está dado de alta.**

13.4.2.4 Cuarto bloque de la etapa 4

En el cuarto bloque de esta segunda etapa codificaremos la funcionalidad *Cancelar una cita.*

Diseño y codificación

En la Figura 13-18 se presenta el diagrama de flujo con el algoritmo de funcionamiento de *Cancelar una cita.*

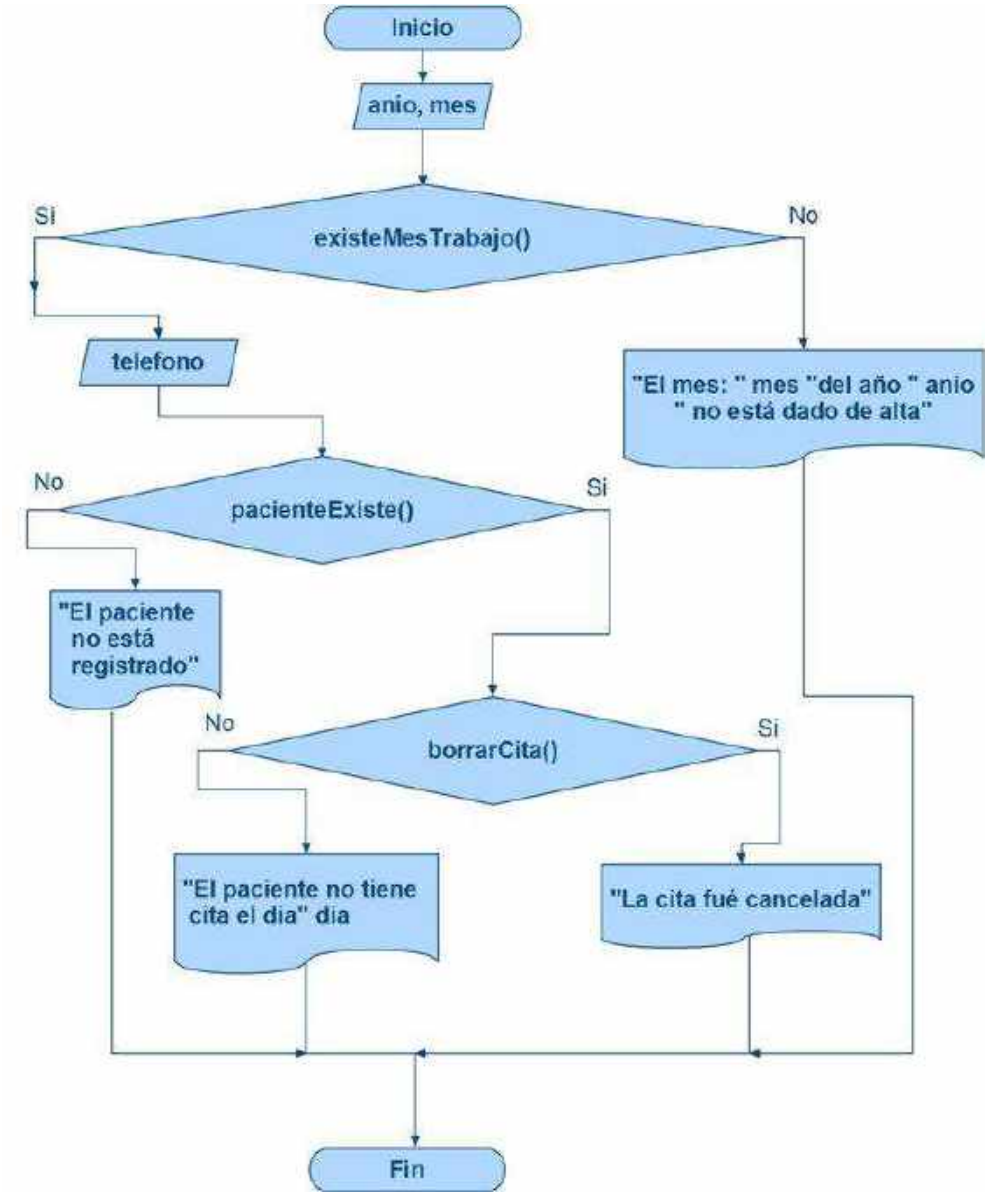


Figura 13-18: Diagrama de flujo de "Cancelar una cita"

Podemos reutilizar el código de

```
private static void registrarCita( ) { . . .}
```

para codificar

```
private static void cancelarCita( ) { . . .}
```

Solo tendremos que hacer algunas modificaciones. Se deja al lector como ejercicio realizar estas modificaciones.

En la Figura 13-19 se presenta el diagrama de flujo con el algoritmo de funcionamiento de *Borrar cita del mes.*

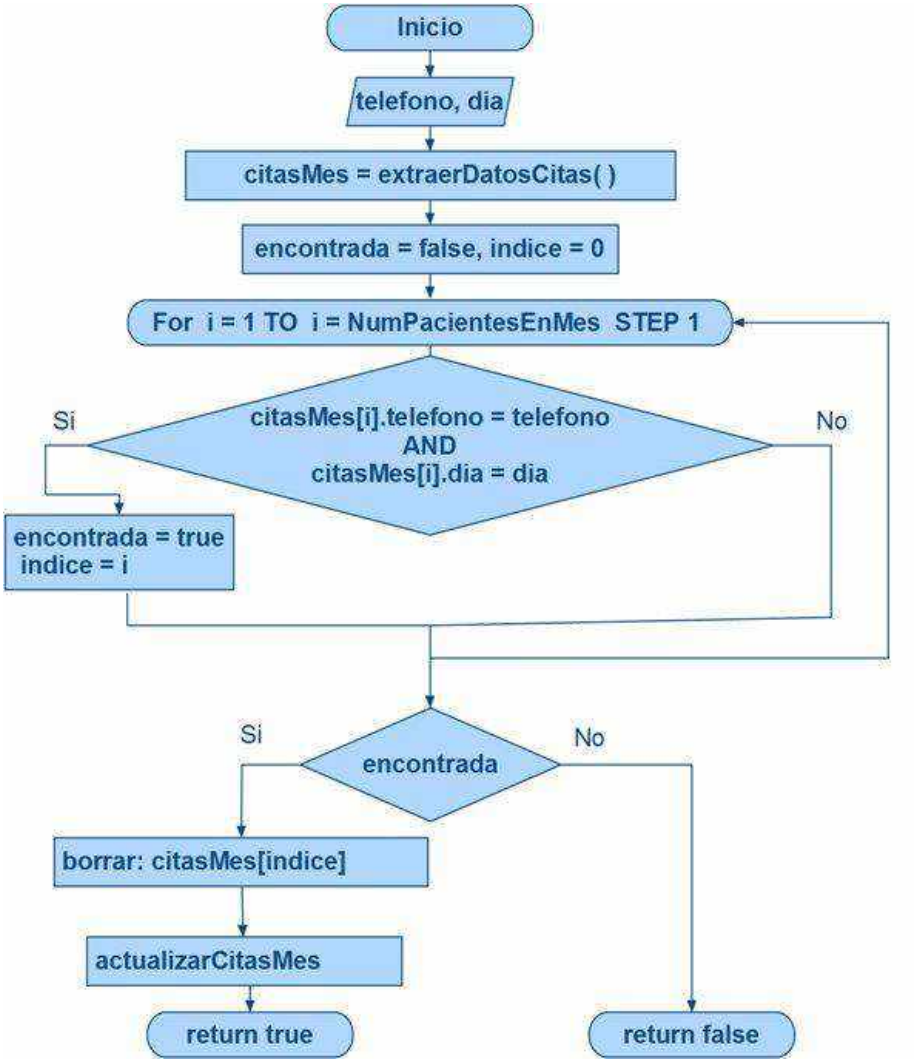


Figura 13-19: Diagrama de flujo de "Borrar cita del mes"

Podemos reutilizar el código de

```
private static void registrarCita( ) { . . .}
private boolean borrarAlumnoDelGrupo( ) {. . .}
private static void actualizarGrupoAlumnos(...){. . .}
```

para codificar

```
private static void borrarCita( ) { . . .}
private static void actualizarCitasMes(...){
```

Solo tendremos que hacer algunas modificaciones. Se deja al lector codificar estos métodos.

En la Figura 13-20 se ilustra la funcionalidad *Cancelar cita* funcionando correctamente.

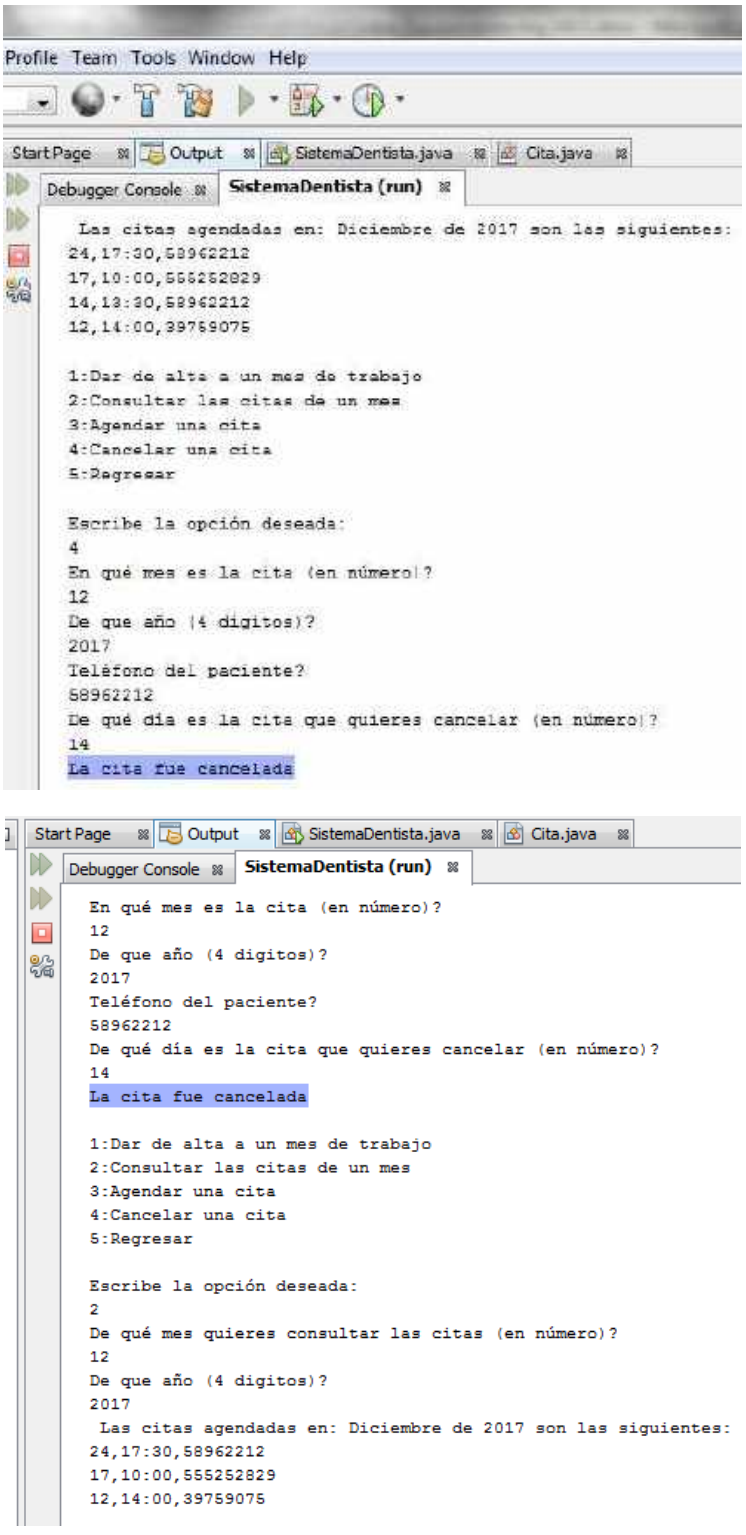


Figura 13-20: Cancelación de una cita

Pruebas de validación

Pruebas del menú citas opción: Cancelar una cita

I.1.- Se selecciona la opción 2 del menú principal. Luego la opción 4 del menú de citas. Se proporciona el mes y el año del mes del que se desea cancelar la cita. El mes de trabajo ya debe estar dado de alta previamente.

Posteriormente se proporciona el teléfono del paciente y el día que tiene agendada su cita, *la cita debe estar agendada*.

El sistema deberá borrar la cita del archivo correspondiente y decir que **la cita fue cancelada.**

I.2.- Se selecciona la opción 2 del menú principal. Luego la opción 4 del menú de citas. Se proporciona el mes y el año del mes del que se desea cancelar la cita. El mes de trabajo ya debe estar dado de alta previamente.

Posteriormente se proporciona el teléfono de un paciente que no está dado de alta

El sistema deberá decir que **el paciente no está registrado.**

I.3.- Se selecciona la opción 2 del menú principal. Luego la opción 4 del menú de citas. Se proporciona el mes y el año del mes del que se desea cancelar la cita. El mes de trabajo ya debe estar dado de alta previamente.

Posteriormente se proporciona el teléfono de un paciente que ya está dado de alta y un *día en el que el paciente no tenga agendada una cita* en ese mes.

El sistema deberá decir que **el paciente no tiene cita ese día.**

I.4.- Se selecciona la opción 2 del menú principal. Luego la opción 4 del menú de citas. Se proporciona un mes o un año del mes que no tenga un mes de trabajo asociado.

El sistema deberá decir que **el mes no está dado de alta.**

14 Modelos y Metodologías Ágiles

14.1 Objetivos específicos del capítulo

- Conocer los principios de los modelos y metodologías ágiles, así como las condiciones necesarias para poder aplicarlos.
- Conocer los modelos y metodologías ágiles más utilizados.
- Conocer las características y limitaciones de los modelos y metodologías ágiles.

14.2 Introducción

Los modelos ágiles son técnicas de desarrollo de software que se basan en un desarrollo iterativo e incremental en ciclos muy cortos. Surgieron a raíz de que los modelos tradicionales no están preparados para hacer frente a los *cambios rápidos en los requerimientos.* Entonces:

El objetivo principal de los modelos ágiles es minimizar el costo de los cambios en los requerimientos.

En 2001 se reunió un grupo de expertos con el fin de mejorar el desarrollo de software. Como resultado elaboraron un documento llamado "Manifiesto para el desarrollo ágil del software" (En inglés: "*The agile manifesto*") [Beck et al., 2001]. En el que se especifican las siguientes ***características de las metodologías ágiles***:

- Dan más valor a los individuos y las interacciones entre ellos que a las herramientas de corrección y gestión de los procesos.

- Se basan principalmente en la experiencia y habilidad de los desarrolladores en lugar de obligarlos a cumplir con ciertas normas.

- Promueven el diseño inicial simple y la comunicación entre las partes interesadas más que una definición formal de las características del sistema requeridas.

- Promueven más la comunicación cara a cara que la documentación completa. (¡Ojo, Esto no significa que no exista la documentación!)

En el Manifiesto Ágil, también se enuncian los siguientes ***aspectos de la filosofía ágil***:

- Es preferible dedicar el tiempo en producir un software que funcione en vez de utilizarlo para producir la documentación.

- Da más énfasis a la colaboración del cliente en los aspectos clave del sistema a desarrollar que la negociación del contrato.

- Es preferible lidiar con los cambios en lugar de desarrollar un plan y seguirlo.

[14]Con la *entrega continua de nuevas versiones* promovida en las metodologías ágiles es más fácil hacer frente a los cambios de última hora en los requerimientos. Las entregas iniciales tienen el objetivo de implementar los requerimientos esenciales del cliente. Y con el uso de las primeras versiones se comprende mejor el problema a solucionar y emergen nuevos requerimientos que son cubiertos en entregas posteriores.

Sin embargo, no en todos los casos es viable aplicar una metodología ágil, ya que deben existir las siguientes ***condiciones para poder aplicarla***:

- Proyectos pequeños y equipos pequeños.
- Requerimientos cambiantes.
- Equipo de desarrollo muy capaz.
- Un cliente dispuesto a participar en el desarrollo del sistema.

[14] https://images.assetsdelivery.com/compings_v2/kentoh/kentoh1506/kentoh150600288.jpg

Historia de usuario.- Es una pequeña descripción simple y clara de un requerimiento del usuario. Está escrita en lenguaje natural y debe ser verificable, es decir, debe tener una prueba asociada. Se escriben en una pequeña hoja autoadherible para tenerla en la pared o en un pizarrón. Cada historia se discute con el usuario para aclararla cada vez más. En la Figura 14-1 se ilustra un ejemplo de historia de usuario.

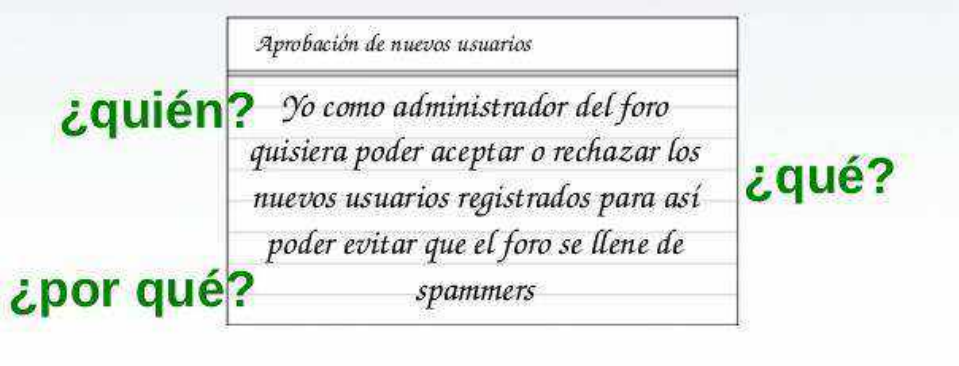


Figura 14-1: Ejemplo de historia de usuario [Universidad de los Andes, Venezuela]

En la Tabla 14-1 se ilustra la clasificación de los modelos y metodologías basadas en la gestión del cambio.

Universal	Modelo Abstracto	Modelo Concreto
Modelos y metodologías basadas en la gestión del cambio (Agiles)	Modelos ágiles con un enfoque de desarrollo y de gestión	Extreme programing (XP) Feature-driven development (FDD) SCRUM Test-driven development (TDD) Dynamic Systems Development (DSDM) Otras
	Metodologías basadas en principios y en prácticas	Agile Modeling (AM) Crystal Lean Otras

Tabla 14-1: Modelos y metodologías ágiles

En las siguientes secciones estudiaremos en qué consiste cada una de ellas.

14.3 Modelos ágiles con un enfoque de desarrollo y de gestión

Los modelos ágiles con un enfoque de desarrollo y de gestión definen una secuencia de proceso específica, así como algunos *aspectos de la gestión* de procesos de software, tales como: tamaño del equipo, duración de la iteración y distribución del equipo de trabajo.

14.3.1 Programación Extrema (eXtreme Programming: XP)

La programación extrema (XP) es un método para equipos pequeños y medianos. XP propone 12 prácticas (reglas) que tienen como objetivo reducir el costo del cambio. La esencia de la programación extrema consiste en la adopción de las mejores metodologías y ciclos de vida de desarrollo de software y aplicarlos de forma dinámica, según las exigencias y características de cada parte (componente, funcionalidad, fase, etc.) del proyecto que se está desarrollando. En otras palabras, la programación extrema propone no adoptar una única metodología o ciclo de vida de desarrollo de software durante todo el desarrollo del proyecto de software, sino que el modelo de desarrollo propuesto cambie de forma dinámica según las características del proyecto. La Figura 14-2 ilustra las fases y la dinámica de la programación extrema.

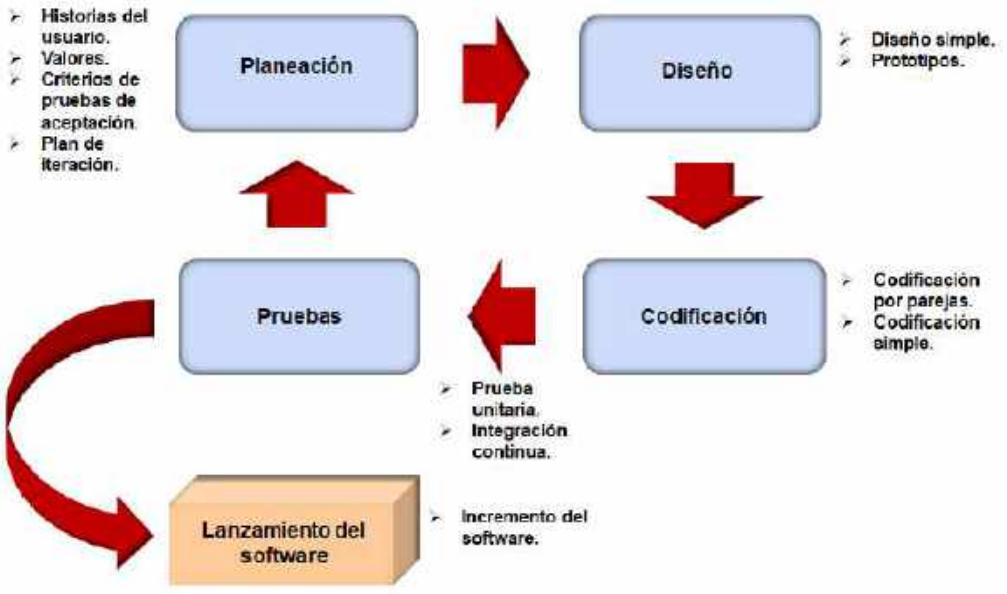


Figura 14-2: Las fases y la dinámica de la programación extrema

La fuerza de la Programación Extrema no es el resultado de cada una de las 12 prácticas, sino de las propiedades emergentes que surgen de su combinación (sinergia). Estas normas pretenden producir diseños sencillos, pruebas automatizadas y una actitud de refinamiento constante del diseño. El ciclo de vida de XP consta de 5 fases: Exploración, Planificación, Iteraciones de Liberación, Producción, Mantenimiento y Muerte.

14.3.2 Desarrollo Dirigido por Características (Feature-driven development: FDD)

El desarrollo dirigido por características (*Feature Driven Development*) comparte varias características con los restantes modelos de desarrollo rápido de software vistos previamente. Este tipo de desarrollo ágil también se caracteriza por su naturaleza incremental. Cada requerimiento es planeado, diseñado, construido y probado de forma individual. Es decir, la planeación, diseño y construcción es por funcionalidad o requerimiento. De esta forma, la metodología de desarrollo del Proceso Unificado, así como la aplicación de los ciclos de vida Espiral, Prototipado evolutivo y Entrega Evolutiva, entre otros, pueden ser vistos como tipos de este desarrollo ágil.

El Desarrollo Dirigido por Características se basa en ciclos de vida cortos, iterativos, basados en características. En primer lugar, se debe hacer un modelo general del sistema con una lista de características informales y notas sobre alternativas. El segundo paso consiste en construir una lista categorizada de características. En la tercera etapa, se elabora un plan de desarrollo teniendo en cuenta la lista categorizada de características. Después se inicia un ciclo iterativo. En cada iteración, se diseña e implementa una característica (construida, probada e integrada). La duración de la iteración es de aproximadamente 1 a 2 semanas. En la Figura 14-3 y Figura 14-4 se ilustran dos perspectivas del proceso del Desarrollo Dirigido por Características.

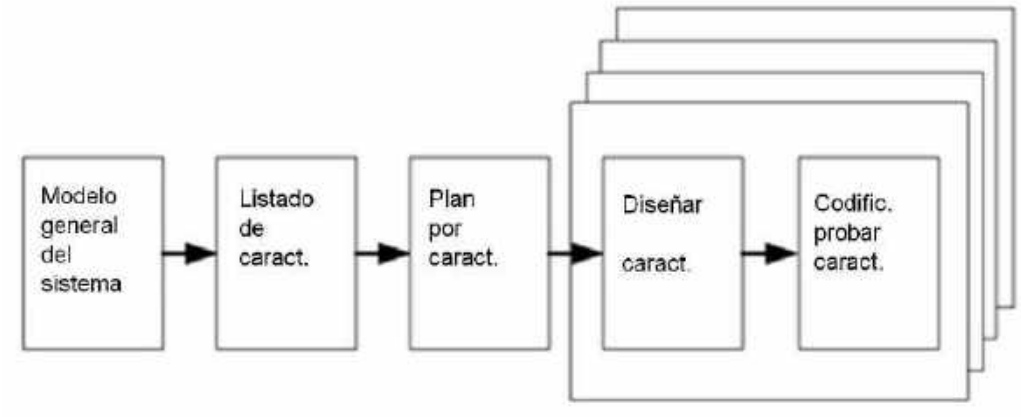


Figura 14-3: Desarrollo Dirigido por Características I

Figura 14-4: Desarrollo Dirigido por Características II

14.3.3 Desarrollo Dirigido por Pruebas (Test Driven Development: TDD)

El Desarrollo Dirigido por Pruebas (TDD: *Test Driven Development*) es un modelo de desarrollo rápido de software que se basa fundamentalmente en dos prácticas de la ingeniería del software:

1. pruebas unitarias de cada requerimiento funcional del software, y
2. refactorización.

Una prueba unitaria es un método que permite comprobar el correcto funcionamiento de un elemento de software (función, procedimiento, clase, componente, etc.), que comúnmente corresponde a un único requerimiento del sistema software que se pretende desarrollar. La refactorización del código fuente se refiere a la reestructuración, modificación o limpieza del código fuente sin afectar su comportamiento. Recordar que la refactorización no corrige errores existentes en el código ni añade nueva funcionalidad al mismo. El principio clave del TDD consiste en primero escribir la prueba unitaria para probar el correcto funcionamiento del requerimiento y después implementar el requerimiento (ver Figura 14-5).

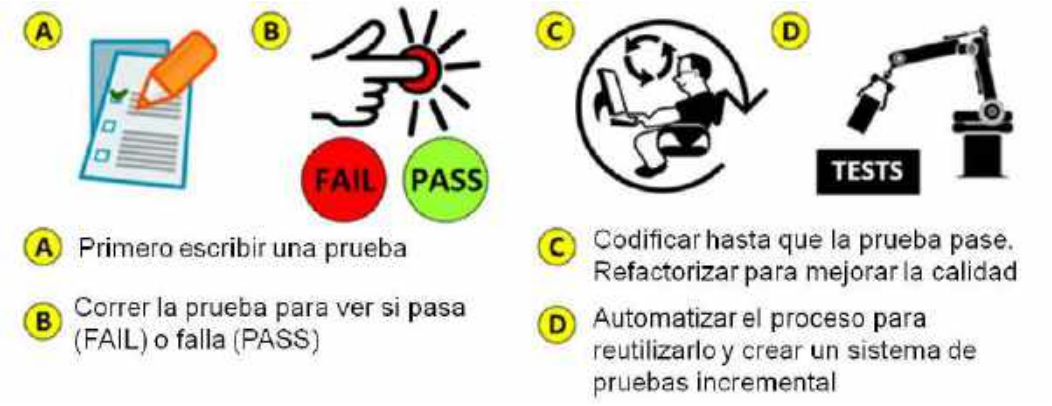


Figura 14-5: El Desarrollo Dirigido por Pruebas[15]

La meta del Desarrollo Dirigido por Pruebas (TDD) es escribir un código limpio que funcione. El TDD consta de tres pasos:

1.- Escribir una prueba automatizada que probablemente no funcionará desde el principio.

2.- Hacer que la prueba funcione rápidamente no importa cómo.

3.- Refactorizar el código para eliminar duplicaciones y mejorar calidad

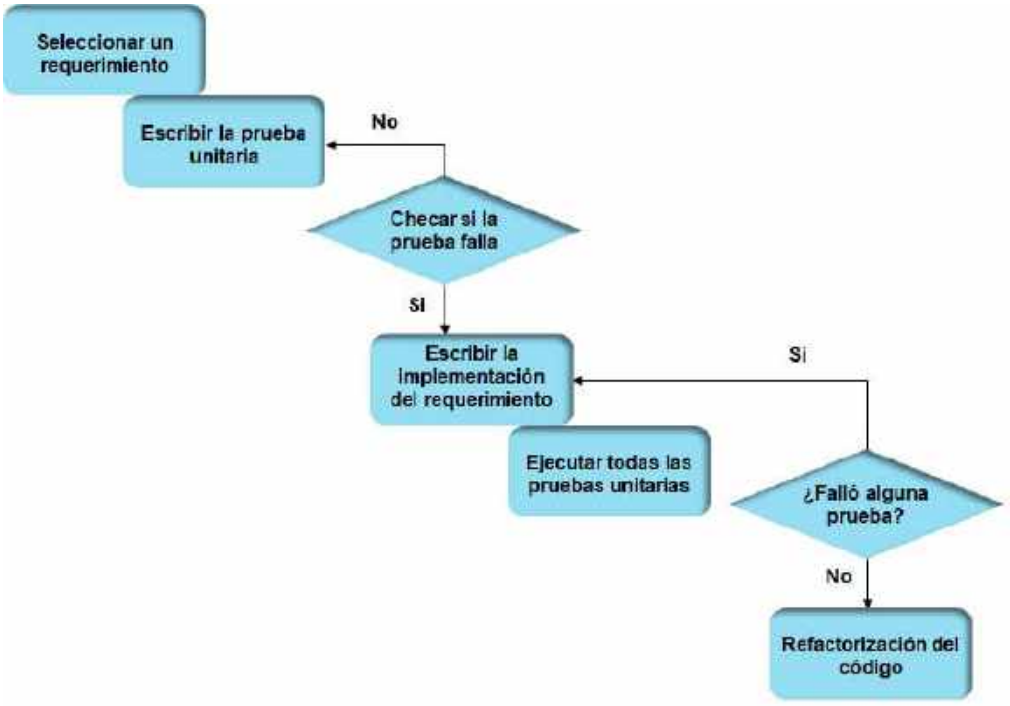


Figura 14-6: Pasos del Desarrollo Dirigido por Pruebas

15 https://lh3.googleusercontent.com/94w9ZXa4qLx1DiiCOOw01qh9bCLNEIaxvpvEI1rp4jXI7nD4GC8Z-RdDu-cJUpgVPE99MA=s170

Los principios de TDD son:

- Reducir la densidad de defectos.
- Reducir las malas sorpresas que aumentan el tiempo de desarrollo.
- Fomentar la comunicación y el trabajo en equipo.

En la Figura 14-6 se ilustran los pasos del Desarrollo Dirigido por Pruebas.

14.3.4 Scrum

[16]*Scrum* es una metodología ágil que se basa en el trabajo eficiente en equipo. Su nombre está inspirado en un tipo de formación de equipo utilizado en el juego de Rugby. *Scrum* es adecuado en caso de requisitos vagos o cambiantes, y se basa en la disponibilidad del cliente para trabajar con el equipo, ya que él es quien elige el orden en el que se implementan las partes del sistema.

Cuando se utiliza un modelo de desarrollo rápido tipo *scrum*, es posible proporcionar signos visibles del progreso, evitando así la percepción de un desarrollo lento por parte del cliente y de la directiva. El modelo *scrum* se ejecuta en fases o iteraciones cortas, cada una de las cuales debe proporcionar un resultado completo, en términos de un incremento de la funcionalidad del prototipo o producto software.

Los proyectos *scrum* se dividen en iteraciones llamadas ***sprints*** que pueden tener de 6 a 30 días de duración (ver Figura 14-7).

[16] https://i.ytimg.com/vi/xwRdc69xKVw/maxresdefault.jpg

Figura 14-7: Un ciclo en scrum se llama Sprint[17]

En la Figura 14-8 se ilustra el funcionamiento del método *scrum*. La *reserva de liberación* (en inglés: *backlog*) es todo el trabajo que se va a realizar para construir el sistema, es decir, son los requerimientos del sistema. Antes de iniciar un *sprint* se hace una "planificación de pre-sprint" en la que se seleccionan las actividades a realizar durante el sprint. Al conjunto de características y funcionalidades que se van a implementar se le llama "*backlog* de *sprint*".

Figura 14-8: Scrum (imagen disponible en internet[18])

En *scrum* se realizan reuniones cortas diarias (15 a 30 minutos) durante el sprint. Durante las reuniones los participantes permanecen de pie, para que no se les ocurra alargar la reunión mientras se echan un cafecito. En cada reunión cada persona contesta tres preguntas:

1.- ¿Qué has completado desde la última reunión?

[17] https://www.lecom.com.br/blog/wp-content/uploads/2018/07/ir_attachment_63-1024x439.png

[18] https://www.lecom.com.br/blog/wp-content/uploads/2018/07/ir_attachment_63-1024x439.png

2.- ¿Qué obstaculizó tu avance?

3.- ¿Qué planeas lograr antes de la próxima reunión?

Después de cada sprint, se lleva a cabo una reunión post-sprint para analizar el progreso del proyecto y demostrar al cliente el sistema actual. El cliente se reúne con el equipo y describe los elementos en la reserva que son los más importantes.

El proceso se repite hasta que se implementa el backlog de *liberación completa*.

Los roles en *scrum*

En *scrum* se les dice los "cerdos" a quienes *están comprometidos* con el proyecto. Este nombre se debe a la siguiente fábula del cerdo y la gallina:

> Una gallina le dice a un cerdo: ¿Por qué no montamos un restaurante?
> El cerdo le responde: Me parece bien, ¿qué nombre le ponemos?
> La gallina, muy resuelta, le contesta: ¿Qué te parece "Huevos con jamón"?
> Tras pensar un breve instante, el cerdo le replica: No me gusta. Tú solo estarías *involucrada* mientras que yo estaría *comprometido*.

Los "cerdos" son: el dueño del producto, el equipo de desarrollo y el scrum Master (el líder del equipo). Los demás stakeholders: clientes, usuarios, ejecutivos, etc., serían las gallinas, ya que solo están involucrados, pero no comprometidos (ver Figura 14-9).

Figura 14-9: Las gallinas están involucradas, mientras que los cerdos están comprometidos (imagen disponible en internet[19])

[19] http://2.bp.blogspot.com/_-OjTFVbeZiE/SuQ0Q9oEFCI/AAAAAAAAANQ/yF-8ioXVXt0/s400/COMPROMETER+vs+INVOLUCRAR.jpg

14.3.5 Método de Desarrollo de Sistemas Dinámicos (Dynamic Systems Development Method: DSDM)

El Método de Desarrollo de Sistemas Dinámicos (DSDM) consta de seis etapas:

1.- Pre-proyecto
2.- Estudio de Factibilidad
3.- Estudio de Negocios
4.- Iteración de Modelos Funcionales
5.- Iteración de Diseño y Construcción
6.- Implementación y Postproyecto.

Su principio fundamental es el siguiente:

En lugar de *fijar la cantidad de funcionalidad* en un producto y luego ajustar el tiempo y los recursos para llegar a esa funcionalidad, se prefiere *fijar el tiempo y los recursos* para después *ajustar la cantidad de funcionalidad* que puede incluirse (ver la Figura 14-10).

Figura 14-10: Método de Desarrollo de Sistemas Dinámicos

El ciclo de vida del DSDM y los detalles del proceso se pueden consultar en: http://www.dsdm.org

14.4 Metodologías basadas en principios y en prácticas

Las metodologías basadas en principios y en prácticas no incluyen una secuencia de proceso específica. Su principal contribución es un conjunto de principios y prácticas que pueden aplicarse a algún proceso específico de desarrollo de software.

14.4.1 Modelado Ágil

El modelado ágil tiene 17 principios que tienen que ver con:

- La entrega frecuente
- La sencillez
- Afrontar los cambios
- Pequeños cambios incrementales
- Utilizar múltiples modelos
- Comunicación honesta y humilde

También sugiere 20 prácticas que promueven principalmente:

- 
[20] La participación activa de las partes interesadas
- La propiedad colectiva de los artefactos (varios tienen acceso para hacer modificaciones)
- Que se pueda probar que cada artefacto es correcto.

[21] El modelado ágil no es un método completo de desarrollo de software. Se centra solo en la documentación y el modelado, y se puede utilizar con cualquier proceso de desarrollo de software. Esta metodología basada en prácticas para la elaboración de modelos y demás artefactos, se ha aplicado con éxito a la Programación Extrema y al Proceso Unificado.

[20] https://www.craiglarman.com/wiki/index.php?title=Agile_Modeling_Pictures

[21] https://gabrielmorrissa.blogspot.com/2013/08/agile-open-medellin-facilitacion.html

14.4.2 Crystal

Crystal son una familia de metodologías de desarrollo de software inventada por Alistair Cockburn (Figura 14-11), quien es uno de los precursores de las metodologías ágiles.


Figura 14-11: Alistair Cockburn es uno de los creadores del "Manifiesto Agil" (imágenes disponibles en internet[22])

[23]Crystal utiliza procesos cíclicos anidados de varias longitudes. El cristal representa una piedra preciosa donde cada faceta es una versión del proceso. Los métodos están codificados por colores para indicar el riesgo para la vida humana. Por ejemplo, los proyectos que pueden implicar riesgo para la vida humana usarán *Crystal Sapphire*, mientras que los proyectos que no tengan tales riesgos usarán *Crystal Clear*.

Crystal se enfoca en seis aspectos principales: personas, interacción, comunidad, comunicación, habilidades y talentos. El proceso se considera secundario. Los métodos son muy flexibles y evitan procesos rígidos debido a que están centrados en la gente. Hay siete propiedades comunes en *Crystal* que indican una mayor posibilidad de éxito y que incluyen:

- La entrega frecuente
- La mejora de la reflexión
- La comunicación osmótica
- Un fácil acceso a los usuarios expertos

La comunicación osmótica de Crystal.- consiste en que el *equipo comparte la misma habitación* (habitualmente conocida como "*sala de guerra*"). Cuando un miembro del equipo hace una pregunta, cualquiera de sus compañeros puede sumarse a la conversación, o bien continuar con su trabajo. De esta forma, la información fluye "por ósmosis" entre los miembros del equipo (ver Figura 14-12).

[22] https://scrumandkanban.co.uk/alistair-cockburn-talk/

[23] https://1.bp.blogspot.com/-44aEa2gSutM/Wkp4sDjX2MI/AAAAAAAAAno/MQs0M6Egeo0FWz5kuaRZu_0gkBKudtWBACLcBGAs/s1600/6.jpg

Figura 14-12: En Crystal el equipo comparte la misma habitación[24]

14.4.3 *Lean* Development

El desarrollo "*lean* (delgado)" es más una filosofía que un proceso de desarrollo. Es una práctica de producción de autos que fue desarrollada por Toyota en los noventas y posteriormente fue adaptada al desarrollo de software.

Si definimos *valor* como aquello que el cliente esté dispuesto a pagar, "*lean*" considera como desperdicio el uso de recursos en actividades que no generen *valor* para el cliente. A continuación, se enlistan los puntos que *lean*, adaptado a la producción de software, considera como desperdicio de recursos y la forma en la que pretende evitar este desperdicio:

Desconocimiento de las expectativas del cliente. Al hacer entregas continuas se obtiene retroalimentación de lo que quiere el cliente.

Duplicar el trabajo. Aunque *lean* fomenta trabajar en pares, la idea es que cada individuo busque defectos desde un punto de vista diferente, minimizando así el tener que volver a hacer el trabajo de nuevo.

---

24 https://www.google.com.mx/imgres?imgurl=http%3A%2F%2Fupscaleresalejackson.com%2Fwp-content%2Fuploads%2F2018%2F11%2Fscrum-office-layout-astonishing-15-best-images-about-scrum-on-pinterest-of-scrum-office-layout.jpg&imgrefurl=http%3A%2F%2Fupscaleresalejackson.com%2Fscrum-office-layout%2Fscrum-office-layout-great-scrum-furniture-by-plan-office-design-by-frans-de-la-haye%2F&docid=PE9jxxsktwbyqM&tbnid=anwyyWRCAVc8oM%3A&vet=10ahUKEwjrrNbSy6XjAhVCVc0KHR4QBdUQMwgpKAAwAA..i&w=403&h=305&itg=1&hl=es-419&bih=613&biw=1366&q=classroom&ved=0ahUKEwjrrNbSy6XjAhVCVc0KHR4QBdUQMwgpKAAwAA&iact=mrc&uact=8

Errores de comunicación. *Lean* fomenta la comunicación verbal como una forma eficiente de obtener claridad.

Inventario incorrecto. A diferencia de metodologías que miden el avance de un proyecto en base a actividades realizadas, en *lean* el avance se mide en términos de "software funcionando".

Errores en el servicio. *Lean* fomenta una retroalimentación continua y la detección de errores lo antes posible a través de revisión y retroalimentación continua.

Potencial humano no utilizado. *Lean* fomenta la participación de todos en todas las etapas de desarrollo y la distribución de roles en forma grupal.

Lean tiene 12 principios que son estrategias de gestión centradas en la satisfacción del cliente y el minimalismo. En la Figura 14-13 se ilustran los principios más sobresalientes de la filosofía *lean*.

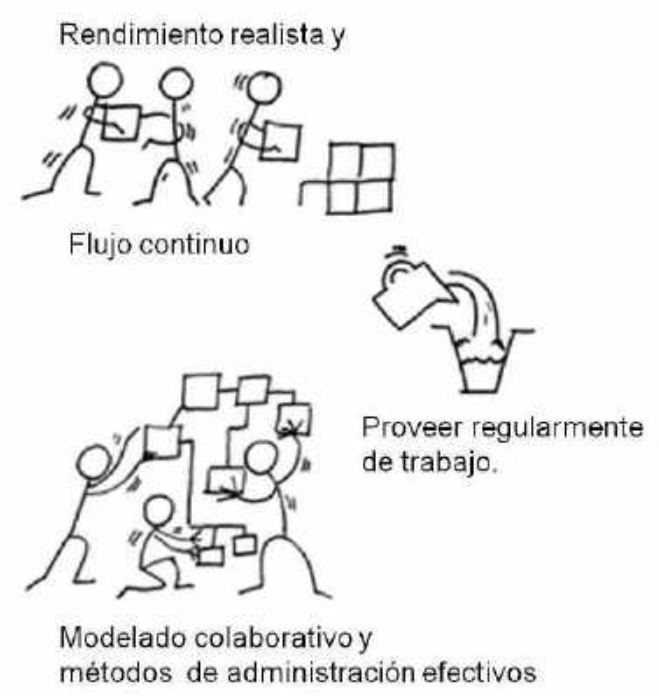


Figura 14-13: Principios de la filosofía Lean[25]

[25] https://lh5.ggpht.com/-Bl_r8W8f2I8/Ue7-NYERfLI/AAAAAAAAEOE/iRmquzwN-Ic/s1600/image%25255B159%25255D.png

14.5 ¿Quién puede participar en el desarrollo ágil?

No cualquier desarrollador es apto para participar en un equipo de desarrollo ágil. A continuación, se presentan las habilidades que debe tener un miembro del equipo en este tipo de desarrollo:

- ***Muy buena capacidad técnica***.- Habilidad para diseñar y programar bien.
- ***Saber comunicarse efectivamente***.- Con claridad, respeto y sin redundancias (evitar el: "…vuelvo a repetir…", que solo quita el tiempo a los demás).
- ***Humildad***.- Significa aceptar que: los demás podrían tener mejores ideas y que las cosas no se tienen que hacer siempre como uno quiere.

14.6 Riesgos del desarrollo ágil

Uno de los riesgos en el desarrollo ágil es la *carencia del documento de Especificación de Requerimientos.* Por ejemplo, en la programación extrema, y en el Desarrollo Dirigido por Pruebas la documentación de los requerimientos se substituye por "casos de prueba" que el sistema debe pasar cuando se implantan ciertos requerimientos. La escasa documentación dificulta hacer cambios al sistema cuando ya se borraron, agregaron o modificaron requerimientos anteriores sobre los que influyen estos nuevos cambios. Si la documentación de los requerimientos en los casos de prueba, es pobre, posiblemente surgirán malos entendidos en su implementación o modificación [Pfleeger y Atlee, 2006].

El desarrollo ágil puede fallar cuando:

- No hay buena comunicación entre los miembros del equipo.
- No se hacen suficientes iteraciones.
- Los equipos son demasiado grandes.
- Los integrantes no están bien capacitados.
- El cliente no hace una retroalimentación efectiva.

14.7 Resumen

Los modelos ágiles son técnicas de desarrollo de software que se basan en un desarrollo iterativo e incremental en ciclos muy cortos. Surgieron a raíz de que los modelos tradicionales no están preparados para hacer frente a los *cambios rápidos en los requerimientos.* El objetivo principal de los modelos ágiles es minimizar el costo de los cambios en los requerimientos.

En las metodologías ágiles se promueve la *entrega continua de nuevas versiones* para poder hacer frente a los cambios de última hora en los requerimientos.

Para poder aplicar una metodología ágil se requiere que los proyectos y los equipos sean pequeños, que el equipo de desarrollo esté formado por desarrolladores capaces y experimentados y que el cliente esté dispuesto a participar en el desarrollo del sistema.

El cuadro de la Figura 14-14 ilustra la clasificación de los modelos ágiles.

Modelo Abstracto	Modelos Concretos
Desarrollo ágil y modelos de enfoque de gestión	Programación extrema SCRUM Desarrollo dirigido por pruebas Desarrollo dirigido por Características Crystal, …, otros
Principios y metodologías basadas en la práctica	Agile, Lean,…, otros

Figura 14-14: Metodologías ágiles

El desarrollo ágil puede fallar cuando no hay buena comunicación entre los miembros del equipo, no se hacen suficientes iteraciones, los equipos son demasiado grandes, los integrantes no están bien capacitados y cuando el cliente no da una buena retroalimentación.

14.8 Cuestionario

1.- ¿Qué son los modelos ágiles?

2.- ¿Cómo logran los modelos ágiles minimizar el costo de los cambios de los requerimientos?

3.- ¿Cuáles son las condiciones para poder aplicar una metodología ágil?

4.- ¿Cuál es la diferencia entre los ***modelos ágiles con un enfoque de desarrollo y de gestión*** y las ***metodologías basadas en principios y en prácticas***?

5.- Relaciona las siguientes columnas. Poner la clave: *número-letra*

1.- *Programación Extrema (XP)*	a) En base a una prueba, se construye el código que hace que ésta sea exitosa.
2.- *Desarrollo Dirigido por Características*	b) Los proyectos se dividen en iteraciones llamadas *sprints* que pueden tener de 6 a 30 días de duración
3.- *Desarrollo Dirigido por Pruebas*	c) Se enfoca en seis aspectos principales: personas, interacción, comunidad, comunicación, habilidades y talentos.
4.- SCRUM	d) Propone la participación activa de las partes interesadas, la propiedad colectiva de los artefactos y que se pueda probar que cada artefacto es correcto.
5.- Modelado ágil	e) Propone 12 prácticas que tienen como objetivo reducir el costo del cambio.
6.- *Crystal*	f) Propone estrategias de gestión centradas en la satisfacción del cliente y el minimalismo.
7.- *Lean*	g) En base a una lista categorizada de funcionalidades se inicia un ciclo iterativo, en cada iteración se construye, prueba a integra una funcionalidad.

6.- ¿Cuáles son las habilidades que debe tener un desarrollador para poder participar en un equipo de desarrollo ágil?

7.- Menciona los casos en los que el uso de metodologías ágiles para el desarrollo de software puede fallar.

15 Bibliografía

- Abrahamsson, P., Salo O., Ronkainen J., Warsta J., (2002), Agile software development methods: Review and analysis, VTT Publications 478.
- Ambler S., (2005), A Manager's Introduction to the Rational Unified Process (RUP), http://www.ambysoft.com/downloads/managersIntroToRUP.pdf.
- Ambler S., Jeffries R., (2002), Agile Modeling: Effective Practices for Extreme Programming and Unified Process, John Wiley and Sons.
- Braude, E. J., (2007) ingeniería de software: una perspectiva orientada a objetos. Alfaomega.
- Beck K., Cockburn A., Jeffries R., Highsmith J, (2001), Agile manifesto, http://www.Agilemanifesto.org.
- Beck K., (2003), Test-driven development: by example, Addison-Wesley Professional.
- Beck K., (1999), Embrace change with extreme programming, IEEE Computer 70–77.
- Booch G., Rumbaugh J., Jacobson I., (1999), The Unified Modeling Language User Guide, Addison-Wesley.
- Bosch J., Stafford J., (2002), Architecting Component-Based Systems, in Building reliable component-based software systems, Chapter 3, pp. 41-55, Artech House Publishers.
- Bourque, P., Dupuis, R., Abran, A., Moore, J. W., Tripp, L., Wolff, S., (2002), Fundamental principles of software engineering–a journey, Journal of Systems and software, 62(1), 59-70.

- Braude, E., (2007), ingeniería de software, una perspectiva orientada a objetos. Editorial Alfaomega.
- Budgen D., Burn A. J., Brereton O. P., Kitchenham B. A., Pretorius R., (2011), Empirical evidence about the UML: a systematic literature review. software: Practice and Experience, 41(4), 363-392.
- Christiansson B., Jakobsson L., Crnkovic I., (2002), "CBD process", in Building reliable component-based software systems, Chapter 5, pp. 89-113, Artech House Publishers.
- Cockburn, A. (2004), Crystal clear: a human-powered methodology for small teams, Pearson Education.
- Crnkovic I., Larsson M., (2005), Building reliable component-based software systems, Artech House Publishers.
- Crnkovic I., Chaudron M., Larsson S., (2006), Component-based development process and component lifecycle, software Engineering Advances, International Conference on IEEE, 44-53.
- Davis, A. M., (1995), 201 principles of software development. McGraw-Hill, Inc.
- Dingsøyr T., Nerur S., Balijepally V., Moe N. B. (2012), A decade of agile methodologies: Towards explaining agile software development, Journal of Systems and software, 85(6), 1213-1221.
- DSDM Consortium, (1997), “Dynamic Systems Development Method”, version 3, Ashford, Eng, DSDM Consortium.
- Fowler, M., (2000), The new methodology, http://www.martinfowler.com/articles/newMethodology.html, [22 - 09 - 2013].
- Fowler, M., Beck, K., (1999), Refactoring: improving the design of existing code, Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., (2003), Patrones de diseño. Addison-Wesley.
- Ghezzi, C., Jazayeri, M., Mandrioli, D., (2002), Fundamentals of software Engineering, second ed., Prentice Hall.
- Gómez M. C., Cervantes J., (2013), User Interface Transition Diagrams for Customer-Developer Communication Improvement in software Development Projects, Journal of Systems and software 86(9), pp. 2394-2410, 2013.
- Grossman, M., Aronson, J. E., McCarthy, R. V., (2005) , Does UML make the grade? Insights from the software development community, Information and software Technology, 47(6), 383-397.

- IBM, Rational Unified Process, Best Practices for software Development Teams, Rational software white paper TP026B, Rev 11/01.
- Maier, Mark W., David Emery, Rich Hilliard (2004), ANSI/IEEE 1471 and systems engineering, *Systems Engineering* 7.3, 257-270.
- McConnell, S., (1997), Desarrollo y gestión de proyectos informáticos, McGraw Hill y Microsoft Press.
- Myers G., (2004), The art of software testing, 2nd edition, John Wiley & Sons Inc.
- Morisio M., Seaman C. B., Basili V. R., Parra A. T., Kraft S. E., Condon, S. E., (2002), "COTS-based software development: Processes and open issues". Journal of Systems and software, 61(3), 189-199.
- Palmer S. R., Felsing M., (2001), A Practical Guide to Feature-Driven Development, Pearson Education.
- Pfleeger S., Atlee J., (2002), software Engineering, Theory and practice, Second edition, Pearson Prentice Hall.
- Pfleeger S., Atlee J., (2006), software Engineering, Theory and practice, Third edition, Pearson Prentice Hall.
- Poppendieck M., Lean programming,

  http://www.Agilealliance.org/articles/articles/LeanProgramming.htm , 2001, [18-10-2013].
- Pressman, Roger S., (2010), ingeniería del software: Un enfoque práctico, 7a ed., McGraw Hill.
- Rising, L., Janoff, N. S. (2000), The Scrum software development process for small teams, software, IEEE, 17(4), 26-32.
- Schwaber, K., Sutherland, J. (2010). What is Scrum. URL: http://www. scrumalliance. org/system/resource/file/275/whatIsScrum. pdf, [Sta nd: 03.03. 2008].
- Sommerville I., (2007), software Engineering, 8ª ed., Addison-Wesley.
- Stephens, Rod, (2015), Beginning software Engineering, John Wiley & Sons.
- Sun Microsystems, Code Conventions for the Java Programming Language.
- softwareEBOK (R), Bourque P., Fairley R. E., Guide to the software engineering body of knowledge: Version 3.0. IEEE Computer Society Press, 2014.
- Tsui F., Karam O., Bernal B., (2014), Essentials of software Engineering, 3a ed, Jonas & Bartlett Learning books.

- Van Vliet H., (2008), software Engineering: Principles and Practice", 3third ed, Wiley & Sons.
- Weitzenfeld, A., (2004), ingeniería de software Orientada a Objetos con UML, Java e Internet, Thomson.
- Wiegers, K., y Beatty, J. (2013). software requirements. Pearson Education.
- Whittaker, J. A., (2002) How to break software: A practical guide to testing. Addison-Wesley.

Fundamentos de Ingeniería de Software
Se terminó de imprimir el 10 de noviembre de 2019 en
Litoprocess S. A. de C.V.
Calzada San Francisco Cuautlalpan 102A
53569 Naucalpan Estado de México.
100 ejemplares en papel bond 90 gr.