UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

"PROCESO DE DESARROLLO DE UNA APLICACIÓN JAVA EMPRESARIAL"

TRABAJO ESCRITO BAJO LA MODALIDAD DE SEMINARIOS Y CURSOS DE ACTUALIZACIÓN Y CAPACITACIÓN PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

PRESENTA:
SANDRA NOEMI ALVAREZ DIAZ

ASESOR: Ing. Enrique García Guzmán

SAN JUAN DE ARAGÓN, EDO. DE MÉXICO 2011.

INDICE

1. INTRODUCCIÓN

Cuando un sistema se desarrolla con éxito y realmente satisface las necesidades de sus usuarios; cuando funciona impecablemente durante largo tiempo; cuando es fácil de modificar e incluso es fácil de utilizar tiene la capacidad de simplificar y minimizar tiempos de ejecución de ciertas tareas. Pero si por el contrario, el software falla, los usuarios no están satisfechos, es propenso a errores, es difícil de modificar o difícil de utilizar, pueden ocurrir verdaderos desastres como el volver a realizar los procesos para los que fue hecho, pero ahora de manera manual.

Al realizar el desarrollo de sistemas todos los involucrados en el proceso desean que el producto realice las tareas de acuerdo a lo planeado evitando que falle ya que si esto ocurre se verán opacados los esfuerzos realizados por el equipo de trabajo. De esta forma, para tener mayor éxito al diseñar y construir un sistema se requiere un enfoque de ingeniería, es decir adaptar modelos de proceso de software, metodologías de desarrollo y herramientas que se puedan ser adaptadas con éxito en el proceso de construcción.

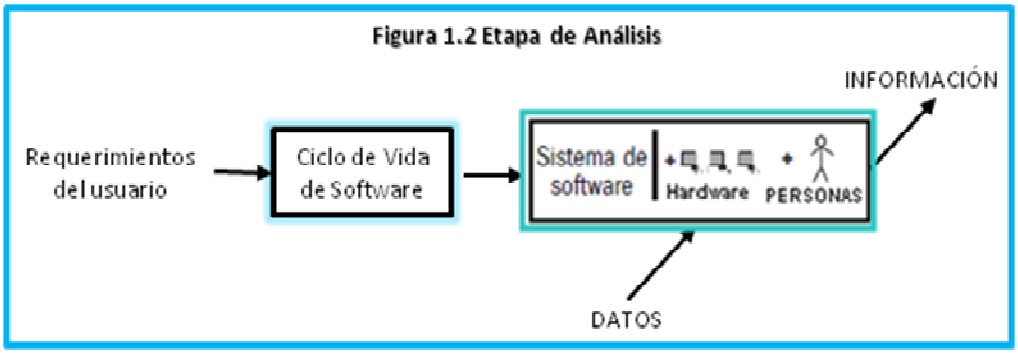
Al elegir una metodología de desarrollo de software, se elige también un modo sistemático para realizar, gestionar y administrar todas y cada una de las etapas de nuestro proyecto, para así de poder realizarlo con altas posibilidades de éxito. La metodología nos indicará cómo dividir el proyecto en módulos pequeños; y las acciones que corresponden en cada una de las etapas definiendo sus entradas y salidas correspondientes, en general nos ayudara a idear, implementar y mantener un producto software, desde que surge la necesidad del sistema, hasta que se cumple el objetivo por el cual fue creado.

Figura 1.1

Existen metodologías análogas a los paradigmas de programación, como la metodología estructurada, que como su nombre lo indica se basa en un paradigma de programación estructurado; o la metodología orientada a objetos que para el tema de estudio de este trabajo nos servirá de marco de referencia. La metodología orientada a objetos arma módulos basados en componentes independientes uno de otro, permitiendo que el código sea reutilizado a largo del desarrollo. (**Ver figura 1.1)**

Cada una de las etapas en el ciclo de vida de un sistema, es fundamental, pero la que marcara de manera decisiva el rumbo y futuro de este, es el Análisis ya que durante el se definen los requerimientos del proyecto, se determinan los elementos que intervienen en el sistema, su estructura general, etc, es decir se obtienen las especificaciones a detalle de lo que se requiere que el sistema haga. **(Ver figura 1.2)**

Figura 1.2 Etapa de Análisis



En la siguiente etapa, que es la Diseño, se debe determinar cómo hacer la construcción del sistema, se definen entidades y relaciones en las Bases de Datos, seleccionamos la tecnología de programación a utilizar, el gestor de Base de Datos, etc. todo a través del modelado del sistema.

Durante la fase de Desarrollo, todo lo que en la etapa anterior se modelo ahora se vuelve código en el lenguaje de programación elegido, en este caso orientado a objetos, se construye la Base de Datos y se implementan las tecnologías necesarias para construir la interfaz de usuario.

Posteriormente en la etapa de Pruebas, se asegura que el sistema desarrollado no falle y que funcione de acuerdo a las especificaciones pactadas desde el Análisis, de esta forma se pueden detectar anomalías e incidentes antes de que el producto sea lanzado a producción.

Una vez que el sistema ha sido probado exhaustivamente y se han corregido todas las desviaciones, es hora de realizar la última etapa dentro del ciclo de vida del sistema que es la Implementación, durante esta fase la aplicación se monta en un servidor publicado en Internet listo para un ambiente productivo es decir, cuenta con datos reales y es capaz de ser utilizado por los usuarios finales.

Es por ello que, a lo largo del presente trabajo se revisará a detalle cada etapa, destacando que no hay metodología que se adapte a la perfección a un desarrollo de software, siempre se deberán elegir las herramientas de acuerdo a la complejidad del problema, el tiempo que se disponga para hacer la entrega del producto, la comunicación que exista entre el equipo de trabajo y el usuario final; resaltando que lo más importante es la certeza que se tenga (o incertidumbre) acerca de que los requerimientos dados por el usuario son completos y correctos.

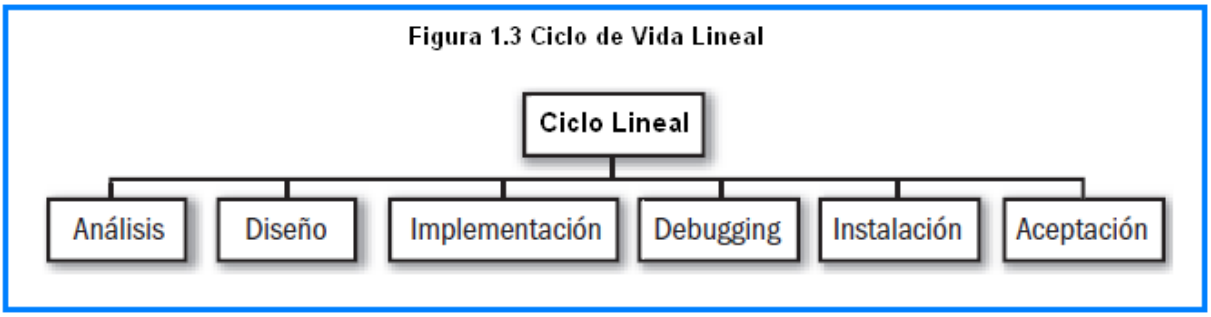
1.1 Ciclo de Vida de un Proyecto

Existen diversos modelos en el ciclo de vida para un proyecto y este se elegirá de acuerdo a las necesidades y alcance del proyecto, de ello también dependerán las etapas en que se dividirá el ciclo de vida, la estructura a seguir, la sucesión de etapas y si se tiene la libertad de realizar repeticiones (iteraciones) entre ellas. Cada modelo a revisar deberá revisar el riesgo que implica su elección, el riesgo propiamente dicho se refiere a la probabilidad de retomar etapas anteriores perdiendo tiempo, dinero y esfuerzo que podrían haber sido empleados para el perfeccionamiento del producto. Los principales modelos de ciclo de vida son:

- **Ciclo de vida lineal:**

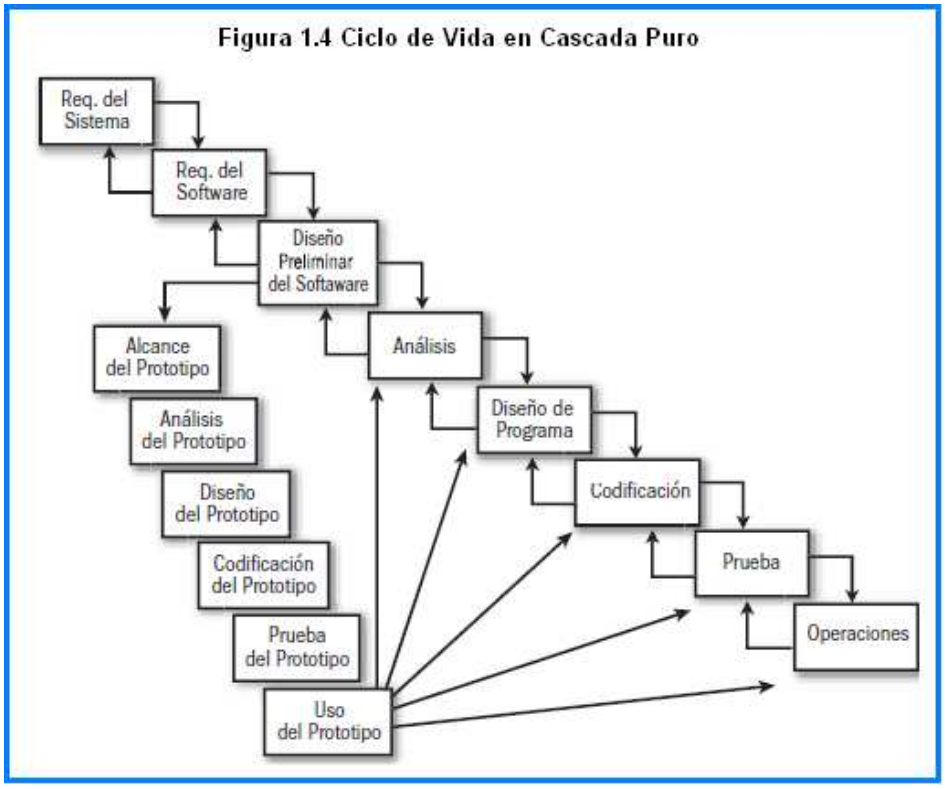
Es el más sencillo de todos, ya que consiste en descomponer el proyecto global en etapas separadas, que se realizarán de manera lineal, es decir que cada etapa se lleva a cabo una sola vez e inmediatamente después de la contigua anterior y no se podrá pasar al siguiente si la primera no ha sido concluida. Los tiempos durante este modelo también son calculados de manera lineal a lo largo de cada etapa.

Figura 1.3 Ciclo de Vida Lineal



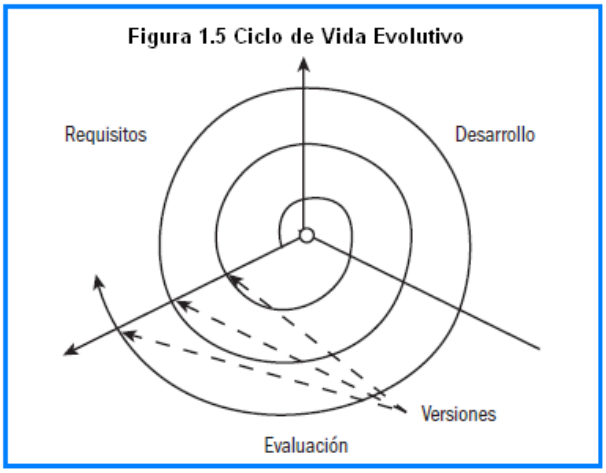
- **Ciclo de vida en Cascada Puro**

Fue propuesto en los 70´s por Winston Royce , en este modelo se admite realizar iteraciones, es decir que al término de cada fase se realizan revisiones para comprobar si realmente se puede pasar a la siguiente etapa; el inconveniente de este modelo es que se requiere contar desde un principio con la idea bastante clara de lo que se requiere desarrollar, ya que si existen errores y no se detectan a tiempo vendrán retrasos que aumenten el tiempo total del proyecto hasta la corrección de los incidentes. **Ver figura 1.4**



- **Ciclo de vida evolutivo:**

Este modelo acepta que los requerimientos del usuario del sistema pueden cambiar en cualquier momento y durante la práctica se demuestra que es sumamente difícil obtener todos los requerimientos necesarios al comienzo del proyecto, debido a que muchas veces el usuario no sabe expresar lo que realmente necesita y aunado a esto que durante el paso del tiempo los requerimientos evolucionan o pueden surgir otros que no fueron planeados. **Ver figura 1.5**

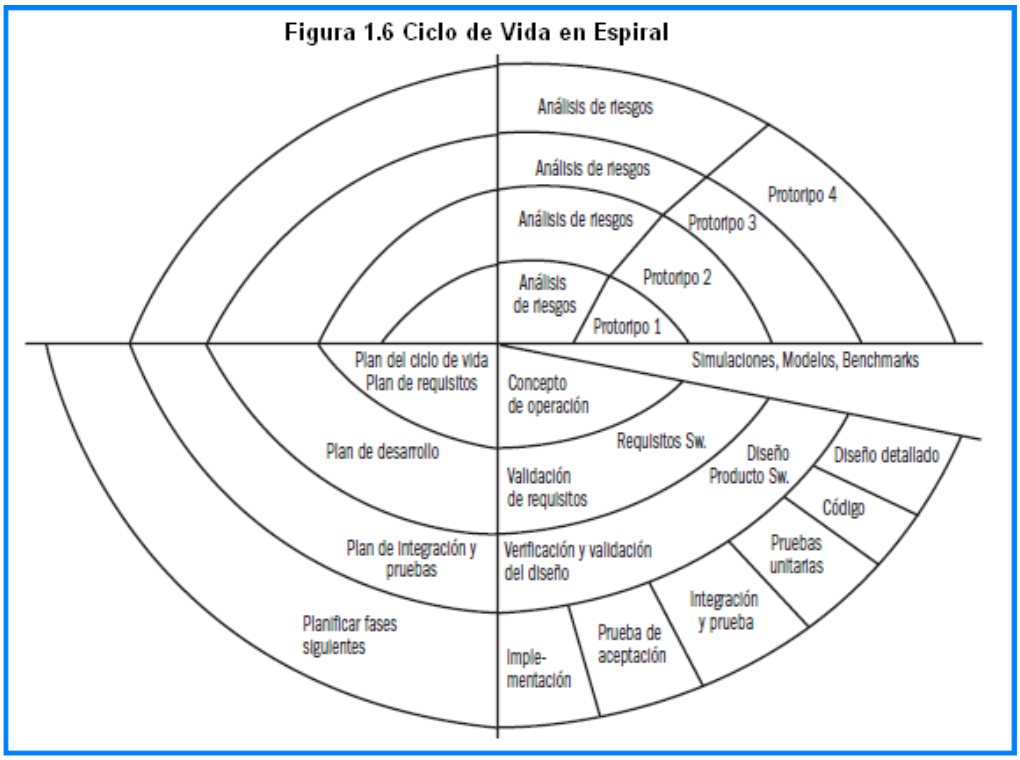


El modelo evolutivo tiene la capacidad de poder afrontar tales situaciones, mediante la iteración de ciclos requerimientos-desarrollo-evaluación, este ciclo de vida resulta ser de gran ayuda cuando no se conocen al 100% los requerimientos iníciales o estos son incompletos, puesto que irán evolucionando a lo largo del desarrollo.

- **Ciclo de Vida en espiral:**

Se basa en una serie de ciclos repetitivos que con el paso del tiempo ganarán madurez hasta llegar al producto final, el modelo maneja el concepto de riesgo, ya que existe cierto grado de incertidumbre acerca de los requerimientos proporcionados en un principio por el usuario y que pueden aumentar a medida que se avanza, durante su desarrollo se obtienen prototipos que el usuario debe validar lo que trae consigo incertidumbre ya que este a veces no sabe que funcionalidades debe tener el producto. Es un modelo empleado en proyectos grandes donde es difícil contar con los requerimientos desde un inicio, y con la repetición de etapas se aumenta la probabilidad de obtener una solución adecuada, momento donde termina el espiral.

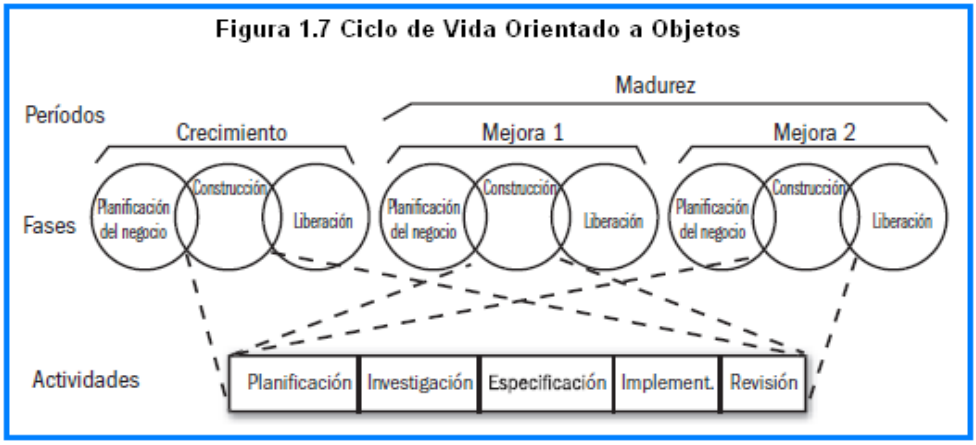
Figura 1.6 Ciclo de Vida en Espiral



- **Ciclo de Vida Orientado a Objetos:**

Modelo introducido durante los 90´s como una de las mejores metodologías a seguir para el desarrollo de sistemas. Al igual que un paradigma de programación orientada a objetos, en este ciclo de vida se utilizan objetos que serán vistos como la funcionalidad o requerimiento solicitado, cada objeto se identificará por un conjunto de propiedades denominadas atributos y al comportamiento del objeto se le nombrará como método, ideas que mantienen el concepto del objeto en casos de la vida cotidiana (abstracción). Se puede emplear independientemente del lenguaje de programación elegido ya que solo es una metodología a seguir y no obliga a la utilización de un lenguaje en particular. **Ver figura 1.7**

Figura 1.7 Ciclo de Vida Orientado a Objetos



Luego de describir algunos de los ciclos de vida más representativos de los sistemas, surge la pregunta de cómo saber cual modelo elegir, la respuesta dependerá de qué tipo de proyecto se pretende desarrollar, que tan amplio es, cuál es la complejidad, el tiempo total que se dispone para desarrollar y lo más importante el tipo de comunicación que exista entre el equipo de desarrollo y el usuario, ya que entre más comunicados estén más detalle se obtendrá en los requerimiento que se deben de plasmar en el sistema.

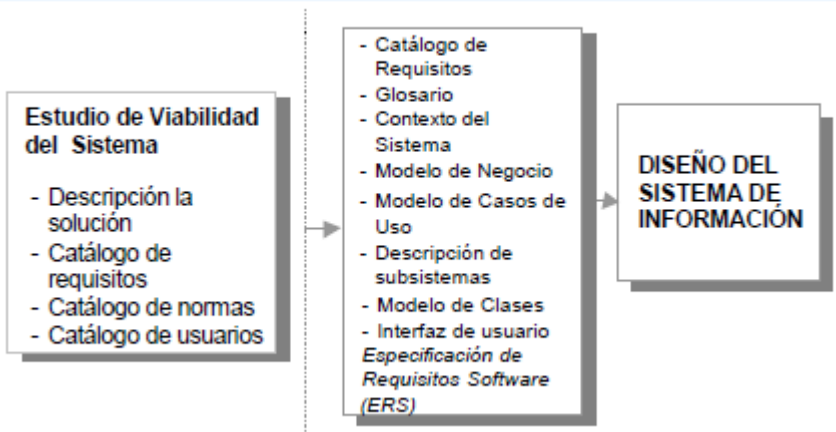
2. ANÁLISIS

Es la primera etapa dentro del ciclo de vida de un sistema, su objetivo es la obtención de una especificación detallada del sistema de información que satisfaga las necesidades de los usuarios y sirva de base para el Diseño del sistema. Se deben de determinar los objetivos, límites del sistema, obtener las características de su funcionamiento y marcar las reglas que permitan alcanzar los objetivos.

En la definición del Sistema, se llevará a cabo una descripción inicial de este, se delimita su alcance, se genera un catálogo de requisitos generales y se describe al sistema mediante modelos iníciales de alto nivel. Después, se definen los Requisitos de la aplicación, para elaborar un catálogo de requisitos, lo que servirá de base para comprobar que la especificación de modelos (diagramas) es completa y soporta los requisitos planteados.

Para obtener Requisitos se toman como punto de partida sesiones de trabajo con usuarios, para reunir la información necesaria y obtener la especificación detallada del sistema. En las entrevistas con el usuario se utilizan Casos de Uso como ayuda y guía en el establecimiento de requisitos, lo que facilita la comunicación con los usuarios y constituye la base de la especificación. Se identifican facilidades que proporcionará el sistema, restricciones de rendimiento, seguridad, control de accesos, etc., incorporándolos al catálogo de requisitos. **Ver Figura 2.1**

Figura 2.1 Elementos de la Etapa de Análisis



a. Definición del sistema:

Se efectúa una descripción del sistema, delimitando su alcance, estableciendo interfaces con otros sistemas e identificando a usuarios representativos. En el caso de análisis orientado a objetos, antes de la captura de requisitos con Casos de Uso, es conveniente establecer el contexto del sistema a partir del modelo de negocio que nos especifica los procesos a los que se quiere dar respuesta en el sistema y el subconjunto de objetos requerido para ello.

b. Identificación del entorno tecnológico:

Se define de manera general, el entorno tecnológico requerido para dar respuesta a las necesidades del sistema, especificando sus posibles condiciones y restricciones, la información se obtiene mediante sesiones de trabajo con usuarios y el apoyo de gente de Tecnologías de Información y Comunicaciones que se considere necesario.

c. Obtención de Requisitos:

Se recoge información de los requisitos que debe cumplir el software y que establecerán los niveles de servicio del sistema, también se definen las prioridades a asignar a los requisitos, considerando los criterios de los usuarios para cubrir las funcionalidades. Los principales tipos de requisitos son:

- Funcionales.
- De Rendimiento.
- De Seguridad.
- De Implantación.
- De Disponibilidad del sistema
- De Casos de uso asociados a los requisitos funcionales

Figura 2.2 Tareas en la obtención de Requisitos

Tarea	Productos	Técnicas y Prácticas
Obtención de Requisitos	- Catálogo de Requisitos - Modelo de Casos de Uso	- Sesiones de Trabajo - Catalogación - Casos de Uso
Especificación de Casos de Uso	- Catálogo de Requisitos - Modelo de Casos de Uso - Especificación de Casos de Uso	- Sesiones de Trabajo - Catalogación - Casos de Uso
Análisis de Requisitos	- Catálogo de Requisitos - Modelo de Casos de Uso - Especificación de Casos de Uso	- Sesiones de Trabajo - Catalogación - Casos de Uso
Validación de Requisitos	- Catálogo de Requisitos - Modelo de Casos de Uso - Especificación de Casos de Uso	- Sesiones de Trabajo - Catalogación - Casos de Uso

d. Análisis de Casos de Uso:

Se identifican clases, cuyos objetos son necesarios para realizar el Caso de Uso y describir su comportamiento a través de su interacción con los objetos. Las clases que se identifican en esta tarea pueden ser:

- **Clase de Entidad**: Representan la información manipulada en el caso de uso.
- **Clase Interfaz de Usuario**: Utilizada para describir la interacción entre el sistema y sus actores, representan ventanas, interfaces, formularios, etc.
- **Clase de Control:** Son responsables de la coordinación, secuencia de transacciones y control de los objetos relacionados con un caso de uso.

e. Validación de Requisitos:

Mediante esta tarea, los usuarios confirman que los requisitos especificados en el catálogo, así como los casos de uso, son válidos, consistentes y completos

2.1 UML como herramienta de modelado

Durante mucho tiempo los desarrolladores de sistemas no realizaban análisis profundos sobre el problema a resolver, comenzaban a diseñar poco al inicio del proyecto y conforme avanzaba el desarrollo se iba analizando, lo que aumentaba el riesgo de no entregar lo que el usuario necesitaba. Hoy en día es necesario realizar un análisis exhaustivo y donde al usuario le quede claro lo que hará el desarrollador.

UML (Lenguaje Unificado de Modelado) es una herramienta que cumple con la función de capturar la idea de un sistema y comunicarla mediante un conjunto de símbolos y diagramas a los involucrados en el proyecto, desde los analistas, hasta los usuarios finales. El objetivo de los diagramas UML es presentar diversas perspectivas de un sistema a las cuales se les conoce como modelo, que describa lo que debe hacer el sistema, pero no dirá cómo debe implementarse en dicho sistema.

UML se compone por elementos gráficos que se combinan para conformar diagramas, la finalidad es presentar diferentes perspectivas del sistema a desarrollar, algunos de los principales modelos son:

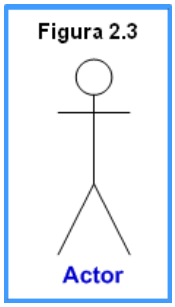
- Diagrama de Caso de Usos
- Diagrama de Clases
- Diagrama de Estados
- Diagrama de Secuencias
- Diagrama de Colaboración

2.1.1 Diagramas de Caso de Uso (CU)

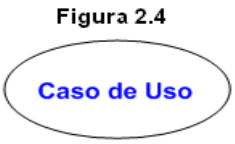
Al adquirir un producto, lo elegimos por sus características, funcionalidades, el tiempo que nos ahorra al facilitarnos ciertas tareas y un sinfín de cualidades. Para elegirlo realizamos un análisis de CU y nos preguntamos cómo lo utilizaremos, de forma que al usarlo satisfaga nuestras necesidades, de a cuerdo a los requerimientos; esto mismo se hace durante el desarrollo de un sistema, pero a través de la etapa de Análisis.

Un CU involucra a los usuarios en la etapa del análisis y diseño del sistema, para poder comprender qué funcionalidad deberá tener, se modelan características del sistema y se desarrollan escenarios. Los CU se componen de:

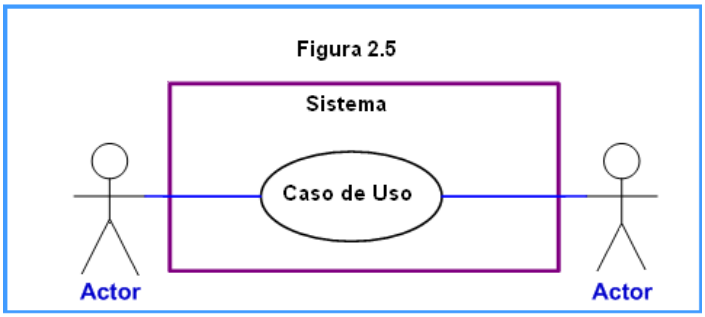
- **Actor:** es una persona, proceso o cosa que interactúa con un sistema. Cada actor participa en uno o varios Casos de Uso. Se simboliza con: **(Ver figura 2.3)**



- **Caso de Uso:** descripción lógica de una funcionalidad del sistema, se representa con una elipse y el nombre del Caso de Uso dentro. **(Ver figura 2.4)**



- **Sistema:** se representa por un rectángulo, con el nombre del sistema dentro de él, dicho rectángulo envuelve los Casos de Uso del sistema y los actores estarán fuera de él como se muestra a continuación. **(Ver figura 2.5)**

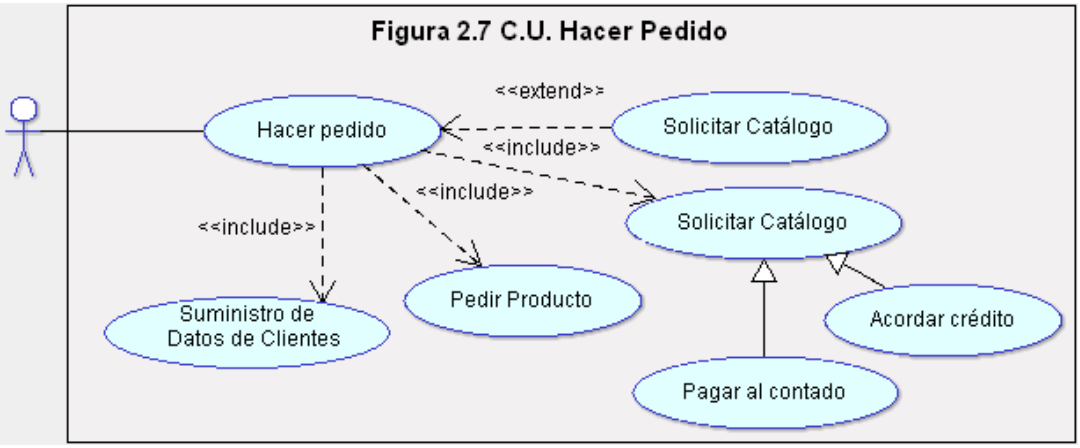


Un CU es una colección de escenarios con una secuencia de pasos a seguir que no aparecerán en el diagrama pero que se adjuntan en notas, como listas de actividades a realizar, actor que inicia el CU, precondiciones de ejecución, pasos del escenario, postcondiones de ejecución, entre otros (Ver Anexo 1). Existen formas en que los Casos de Uso se relacionan entre sí: **(Ver figura 2.7)**

- **Inclusión <<include>>:** indica que se pueden volver a utilizar los pasos de un Caso de Uso dentro de otro. **(Ver figura 2.6)**

- **Extensión <<extend>>:** permite crear un Caso de Uso a través de adicionar pasos a un Caso ya existente. **(Ver figura 2.6)**

Figura 2.6 Tipos de Relciones en un Caso de Uso		
Relación	*Función*	*Notación*
asociación	La línea de comunicación entre un actor y un caso de uso en el que participa	———
extensión	La inserción de comportamiento adicional en un caso de uso base que no tiene conocimiento sobre él	«extend» - - - →
generalización de casos de uso	Una relación entre un caso de uso general y un caso de uso más específico, que hereda y añade propiedades a aquél	——▷
inclusión	Inserción de comportamiento adicional en un caso de uso base, que describe explícitamente la inserción	«include» - - - →



2.1.2 Diagrama de Estados.

Tanto analistas, diseñadores y desarrolladores, deben saber la forma en que los objetos debe comportarse, y en que tendrán que establecer en el código; ya que no es suficiente con implementar un objeto, sino saber lo que realmente debe de hacer. Un diagrama de Estados, es una manera de ejemplificar los cambios en un sistema, es decir, muestra cómo los objetos que lo componen modifican su estado en respuesta a los sucesos y al tiempo.

Además ilustran qué eventos pueden cambiar el estado de los objetos, normalmente contienen: estados y transiciones y estos a su vez, eventos, acciones y actividades. Al igual que otros, estos pueden contener notas explicativas y restricciones.

Evento: Ocurrencia causante de la transición de un estado a otro de objetos, como:

- Condición que toma valor verdadero (expresión booleana). Evento Cambio.
- Recepción de una señal de un objeto a otro. Evento Señal.
- Recepción de una llamada a una operación. Evento Llamada.
- Paso de un período de tiempo o de hora y fecha concretas. Evento Tiempo.

Acción: Operación elemental, que no se puede interrumpir por un evento y se ejecuta hasta su finalización, estas pueden ser:

- Una llamada a una operación
- La creación o la destrucción de un objeto,
- El envío de una señal a un objeto.

Actividades: Mientras un objeto está en un estado, dicho objeto realiza un trabajo que continuará hasta que sea interrumpido por un evento. Por lo que, una acción contrasta con una actividad, ya que ésta puede ser interrumpida por otros eventos.

Estados: Identifica una condición o una situación en la vida de un objeto durante la cual satisface alguna condición, ejecuta alguna actividad o espera que suceda algún evento. Se representa por un rectángulo con bordes redondeados y con tres divisiones internas que alojan nombre del estado, valor característico de atributos del objeto y las acciones que se realizan en ese estado, respectivamente. **(Figura 2.8)**

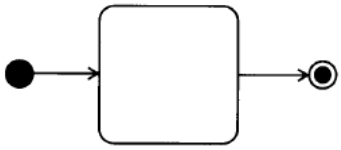
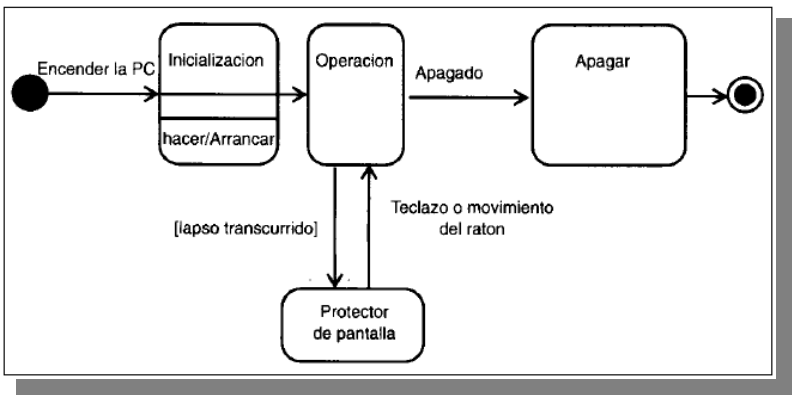


Figura 2.8 Representación de Estados

Por ejemplo: (consultar el Anexo 2)

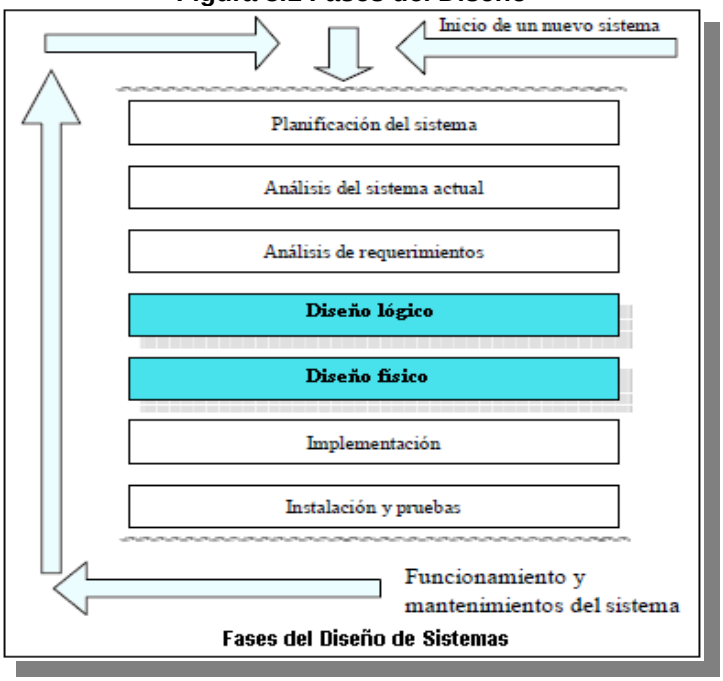


3. DISEÑO

En las fases que forman el Análisis de sistemas, los desarrolladores y los usuarios han definido qué requerimientos funcionales y no funcionales debe de cumplir el sistema. A partir de aquí, el analista y el diseñador, con colaboración de usuarios, diseñan una solución que convierta los requerimientos en un sistema real.

Durante el Diseño los esfuerzos se centran en cómo realizar el sistema, investigando qué datos se van a almacenar, cómo se deben almacenar, qué procesos y cómo se deben implementar, interfaces que se requieren, entre otras. La etapa de Diseño se compone de dos fases, una de Diseño Lógico y otra de Diseño Físico. **(Ver figura 3.1)**

Figura 3.1 Fases del Diseño



Ambas fases deben realizarse de manera secuencial, es decir, la segunda no puede empezar sin que la primera haya finalizado, ya que el modelo Lógico se centra definir qué funciones lógicas plasmadas en los Casos de Uso que se implementan sin tomar en cuenta la tecnología; y el modelo Físico describe qué tecnología (software y hardware) emplear y cómo emplearla para dar solución al Análisis; por ello no se puede empezar el diseño Físico sin terminar el diseño Lógico.

3.1 Diseño Lógico

En esta etapa, se traducen los escenarios de Casos de uso (del capítulo anterior), en un conjunto de objetos de negocio y servicios, ya que estos, son parte de la especificación funcional que se usará para el diseño físico, mientras que el diseño lógico es independiente de la tecnología y comprende lo siguiente:

Objeto de negocios: encapsulación de servicios abstrayendo sus cualidades esenciales.

a. Los objetos de negocio deben verificarse y probarse, asegurando que los módulos operen como unidades completas de trabajo e incluyen:

- Verificación independiente:
  - Pre y post condiciones
  - Lógica y funcionalidad individual

- Verificación dependiente:
  - Verificación de dependencias
  - Que operan como una unidad específica de trabajo

Para definir los objetos de negocios y sus servicios se puede usar la técnica de análisis nombre-verbo de los escenarios de uso. También se puede emplear la técnica sujeto-verbo-objeto directo. En estas técnicas el sujeto y el objeto directo son los candidatos a objetos de negocio y los verbos son los candidatos a servicios.

b. Una Interface puede componerse de las siguientes partes:

- Nombre
- Precondiciones, lo que debe estar presente antes de ejecutarse
- Postcondiciones, estado final
- Capacidad o funcionalidad (SQL, pseudocódigo, función matemática)
- Dependencias

c. La tarea de identificar las dependencias entre objetos permite detectar eventos, sucesos o condiciones que permitan la realización de tareas de negocio coordinadamente, para lo cual se debe considerar lo siguiente:

- Determinar cualquier dependencia (existencial o funcional)
- Determinar cualquier problema de consistencia o secuencia
- Identificar cualquier regulación de tiempo crítica
- Identificar y auditar los requerimientos de control
- Determinar lugares y dependencias a través de la ubicación
- Determinar cuando el servicio que controla la transacción es dependiente de los servicios contenidos en otros objetos de negocio

d. Durante la validación del modelo lógico debe revisar lo siguiente:

- **Completo:** debe representar todos los escenarios de uso.
- **Correcto:** comportamiento lógico correspondiente con el conceptual.
- **Claro:** los objetos de negocio y servicios no deben ser ambiguos.

También el diseño lógico se divide en tres niveles de servicios, que ayudan a tener una aplicación flexible ante cambios de requerimientos, modificando solo las capas necesarias: **(Ver figura 3.2)**

1. **Servicios de usuario:** Controlan la interacción, generalmente involucra interfaces gráficas de usuario (GUI) o no visuales (mensajes o funciones), maneja los aspectos de la interacción con la aplicación; para llegar a minimizar esfuerzos de conocimiento para interpretar la información.

2. **Servicios de datos:** Servicios de bajo nivel que apoyan servicios de negocio y son de una amplia gama de categorías como:

   - Declarar un esquema y su evolución (estructuras de datos, tipos, acceso)
   - Respaldo y recuperación (recuperación de datos si un evento falla)
   - Búsqueda y Lectura (compilación, optimización y ejecución de datos)
   - Inserción, actualización y borrado (procesar modificaciones).
   - Validación de datos (verifica la integridad de estos antes de aceptarlos)
   - Seguridad (acceso a objetos, operaciones, usuario, grupos o servicios)
   - Distribución de datos (a múltiples unidades de recuperación).

3. **Servicios de negocio:** convierten datos recibidos del punto 1 y 2 (servicios de datos y usuario). Las tareas de los servicios de negocio son:

   - Dar formato a los datos
   - Obtener y mover datos desde y hasta los servicios de datos
   - Transformar los datos en información
   - Validar datos inmediatamente en el contexto o terminada la transacción.

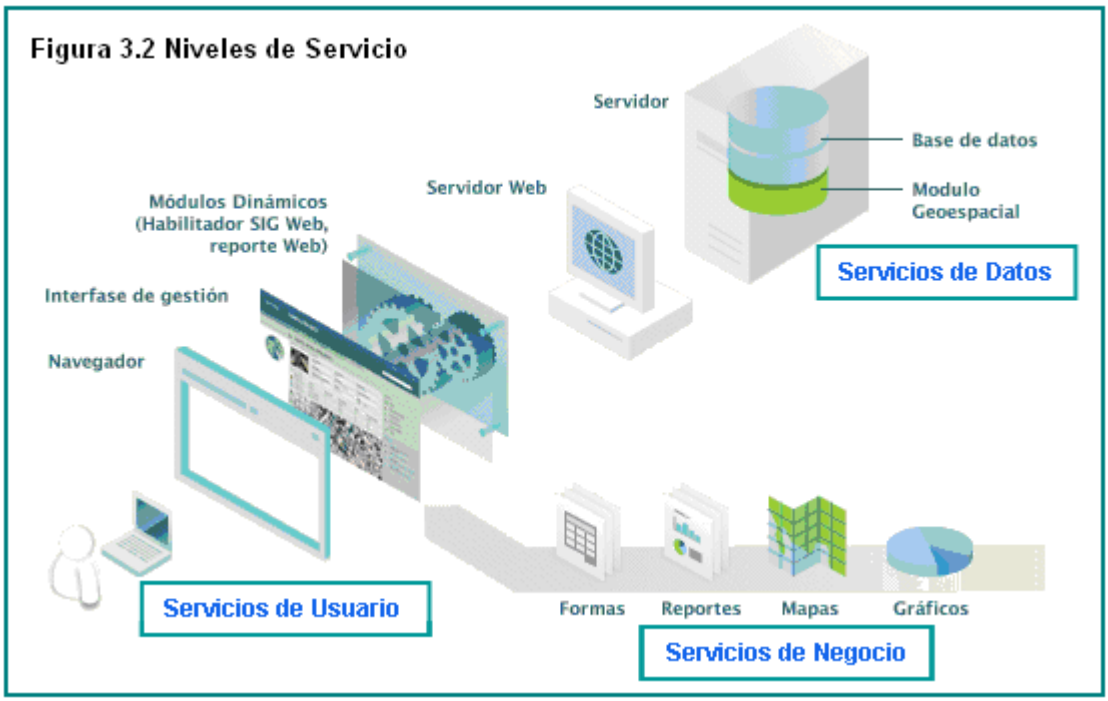


Figura 3.2 Niveles de Servicio

3.1.1 Diagrama de Clases

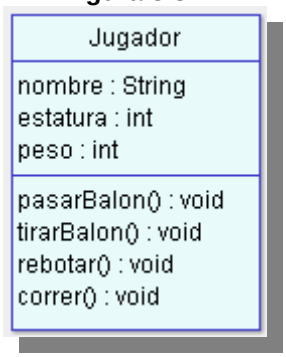
Al pensar en cosas que nos rodean, pensamos también en sus atributos (propiedades) y en las acciones que realizan, entonces encontramos que las cosas debido a su propia naturaleza pueden agruparse en categorías (automóviles, muebles, electrodomésticos, frutas, etc.) dichas categorías se denominan Clases.

Una Clase es una categoría o grupo de cosas que tienen atributos y acciones similares, un ejemplo muy claro es la Clase automóvil, que tiene atributos tales como marca, modelo, número de serie, número de puertas, cilindros, etc.

Simbología: La clase se representa con un rectángulo, por convención, el nombre de una Clase comienza con mayúscula y se coloca en la parte superior del rectángulo, si el nombre de la Clase se compone de dos palabras, se pueden unir al juntarlas y cada una irá con la primer letra mayúscula, o con guión bajo. **(Figura 3.3)**

a. **Intermedio:** Contiene atributos (variables) que caracterizan la Clase.
b. **Inferior**: Contiene métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno.

Figura 3.3



c. **Atributos y Métodos**:

- **Atributos:** Propiedades o características de una Clase, por convención se escriben en minúsculas, pero cuando se componen de dos o más palabras la segunda ira sin espacio y comenzando con mayúscula. Los atributos pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son: public, prívate o protected.

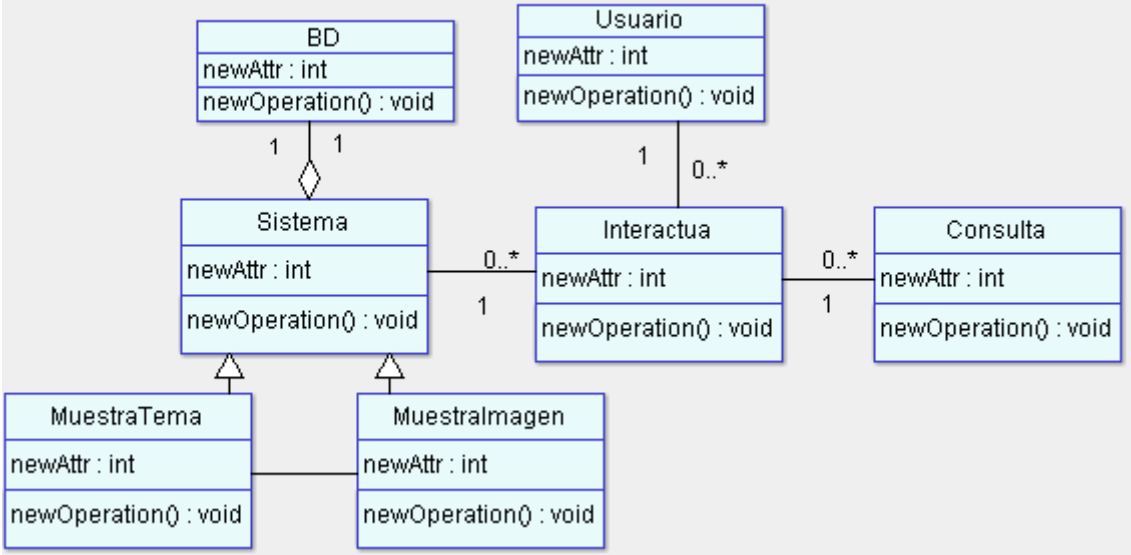
- **Métodos:** Operaciones de una clase son la forma en cómo ésta interactúa con su entorno, éstos pueden tener las características:

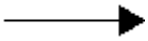
  - **public**: El método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
  - **private**: El método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).
  - **protected**: El método no será accesible fuera de la clase, pero si podrá ser accedido por métodos de su clase y por métodos de subclases que deriven

d. Relaciones entre Clases: Dos o más Clases se pueden interrelacionar, cada una con características y objetivos diferentes, como a continuación se muestra:

- **Cardinalidad:** propiedad que indica el grado y nivel de dependencia de las Clases, se anotan en cada extremo de la relación como: **(ver figura 3.4)**
  - **uno o muchos:** 1..* (1..n)
  - **0 o muchos:** 0..* (0..n)
  - **número fijo**: m (m denota el número).

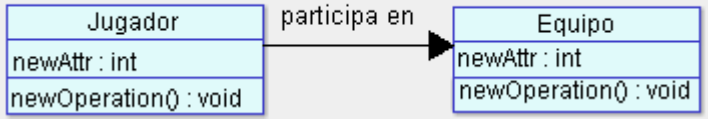
Figura 3. 4 Cardinalidad

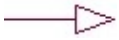


- **Asociación**: 

La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro. Se representa con una línea que conecta ambas Clases colocando el nombre de la asociación sobre la línea, además se debe indicar la dirección de la asociación mediante un triangulo relleno al final de la línea; por ejemplo: **(figura 3.5)**

Figura 3.5 Asociaciones

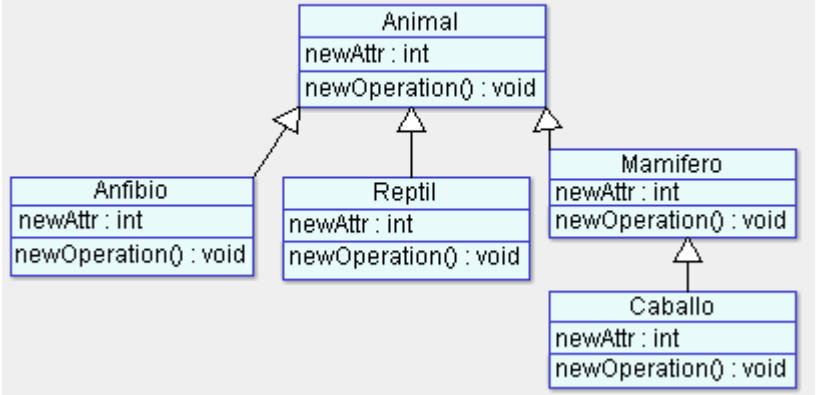


- **Herencia (Generalización)**: 

Generalización: Indica que una subclase hereda métodos y atributos especificados por una Super Clase (Clase Padre) y por ende la Subclase además de poseer métodos y atributos propios, posee características y atributos de la Super Clase, por ejemplo un mamífero es una Clase secundaria, de Animal y Caballo es una Clase secundaria de Mamífero.

La herencia se representa por una línea que conecte la Clase principal con la secundaria, al final de la línea se coloca un triangulo sin rellenar que apunte a la Clase principal; esto se interpreta como: Mamífero **es un tipo de** Animal y Caballo **es un tipo de** mamífero.

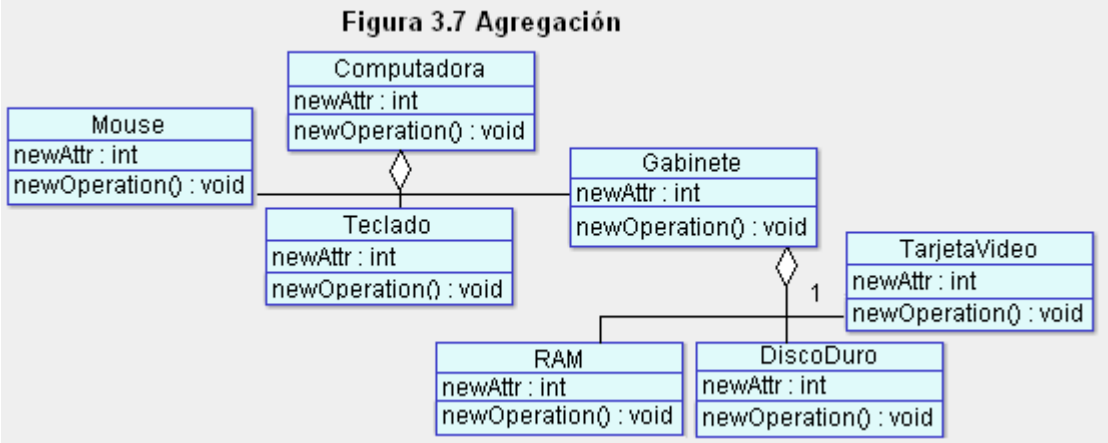
Figutra 3.6 Generalización



- **Agregación**: 

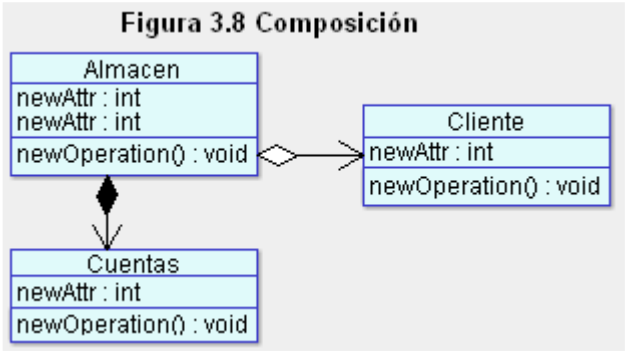
En ocasiones, una Clase se compone de otras Clases, a esta relación se le conoce como Agregación; se puede representar como una jerarquía dentro de la Clase completa por ejemplo una computadora, en la parte superior y en la inferior sus componentes. Se representa por una línea que conecta el todo con un componente y con un rombo sin relleno que se coloca en la línea más cercana al todo en este caso la computadora (figura 3.7):

Figura 3.7 Agregación



- **Composición:** Tipo específico de agregación en donde cada componente dentro de la composición puede pertenecer tan sólo a un todo, se representa con un rombo relleno (Ver figura 3.8). Cada Clase juega un papel y su multiplicidad especifica cuántos objetos de esta se relacionan con un objeto de la Clase asociada; una asociación puede contener atributos y operaciones.

Figura 3.8 Composición



Una Clase puede heredar atributos y operaciones de otra Clase, la clase heredada es secundaria de la Clase principal, esta propiedad se descubre cuando se observa que las Clases del modelo tienen atributos y operaciones en común.

3.1.2 Diagrama de Secuencias

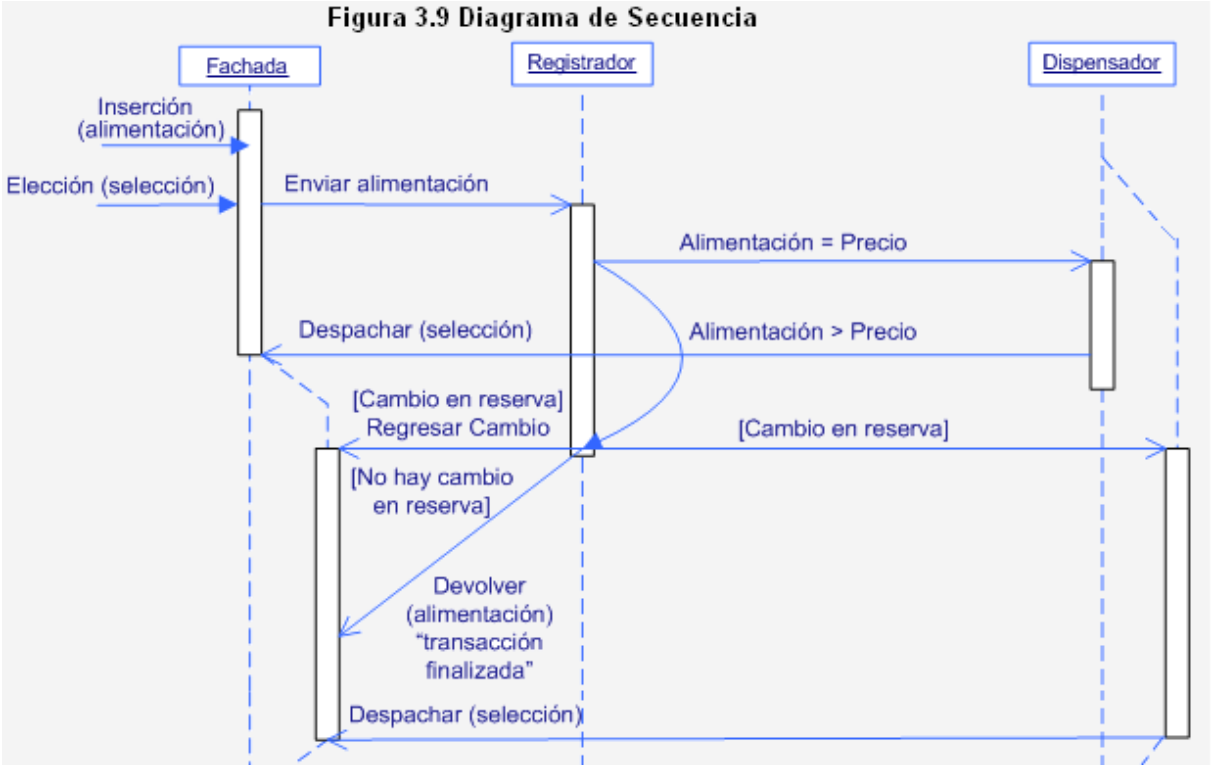
Un diagrama de secuencia, muestra la mecánica de interacción entre objetos a través del tiempo, es decir cómo se comunican los objetos entre sí, mediante interfaces, para poder invocar a una operación. Proporciona un camino a partir de los escenarios de los CU, para describir las operaciones en una forma más detallada. El diagrama se modela a nivel de objetos y utiliza tres elementos fundamentales: objetos, mensajes/estímulos y líneas de vida de los objetos. A continuación algunos puntos importantes a considerar en los diagramas de este tipo:

- El primer mensaje de un diagrama de secuencia siempre inicia arriba del lado izquierdo del diagrama, los siguientes aumentan ligeramente más abajo.
- Para mostrar un objeto (línea de vida) que manda un mensaje a otro objeto, se usa una línea con una punta de flecha rellena (mensaje que requiere respuesta), una flecha simple (mensaje simple) o flecha a la mitad.
- El mensaje (nombre del método) se coloca arriba de la flecha, este representa una operación/método que la clase objeto receptora va a implementar.

Escenario: comprar una lata de refresco depositando una cantidad incorrecta en la máquina expendedora. **(Figura 3.9)**

1. El registrador de la máquina verifica si el importe del cliente concuerda con el precio del refresco.
2. Si el monto es mayor al precio, el registrador de la máquina calcula el cambio a regresar y verifica si tiene cambio.
3. Si cuenta con cambio, la máquina devuelve el cambio al cliente y continúa la entrega del refresco.
4. Si no existe cambio, la máquina regresa el monto depositado y muestra un mensaje para que se deposite el monto exacto.
5. Si la cantidad insertada es menor a la indicada, la máquina no hace nada y espera a que se deposite el resto.

Figura 3.9 Diagrama de Secuencia



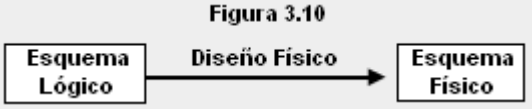
El diagrama de secuencia, sirve para descubrir interfaces requeridas en cada objeto y validar que cada una se usa como debe de ser. Además ayuda a modelar interacciones entre objetos, que pueden resultar complejas, se modela un juego de interacciones como un escenario, tomando en cuenta lo siguiente:

- Identificar los objetos participantes.
- Dibujar una línea vertical bajo cada objeto, que representa su línea de tiempo.
- Los mensajes son una línea horizontal del objeto que manda al que recibe.
- Para un mensaje síncrono llamada se requiere una respuesta, para los asíncronos no se requiere respuesta.

3.2 Diseño Físico

El diseño físico de un sistema, es la forma en que se lograrán las tareas que realizará el sistema, incluye la manera de conjuntar sus componentes y las funciones que realizará cada uno, se especifican las características de los componentes requeridos para poner en práctica el diseño lógico. A continuación **(figura 3.10)** se enlistan algunos de los componentes que se deben delinear:

Figura 3.10



- **Diseño de hardware:** debe especificarse el equipo de cómputo, que incluye dispositivos de entrada, procesamiento y salida, junto con sus características de rendimiento, por ejemplo se requerirá el diseño de la Base de Datos y especificar los dispositivos de almacenamiento con la capacidad suficiente para almacenar grandes volúmenes de datos si se necesita.

- **Diseño de Software:** deben especificarse las características del software, tal es el caso de un sistema administrador de Bases de Datos en algunos casos se puede desarrollar y en otros simplemente se adquiere mediante un licencia, además se especifica la capacidad de acceder a los datos almacenados en ciertos archivos dentro del sistema.

- **Diseño de la Base de Datos:** es necesario detallar el tipo, estructura y funciones de la Base de Datos, relaciones entre los elementos establecidas durante el diseño lógico, los accesos incluyen aspectos como las rutas de acceso y la organización de la estructura de archivos. **(figura 3.11)**

Figura 3.11 Diseño de BD

- **Diseño de telecomunicaciones:** deben especificarse características necesarias del software, medios, dispositivos de tele comunicaciones, de tal forma que se puedan compartir datos y software, configurando la red local y el software de telecomunicaciones necesarios.

3.2.1 Modelo Entidad-Relación (E-R)

Este modelo sirve para crear esquemas conceptuales de BD y es prácticamente un estándar para crear esta tarea. Se compone de los siguientes elementos:

Entidad: cualquier objeto o elemento (real o abstracto) del que se pueda almacenar información en la BD, es un objeto que posee múltiples propiedades (atributos), se representa por medio de rectángulos con su nombre escrito en el interior, el nombre de una entidad solo puede aparecer una sola vez en el modelo.

Figura 3.12 Elementos del Diagrama E-R

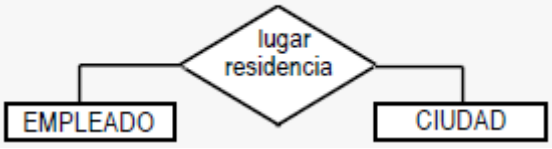
Tipos de entidades:

- **Regulares.** Entidades normales que existen por sí mismas sin depender de otras.

- **Débiles.** Su existencia depende de otras. Por ejemplo la entidad **tarea laboral** sólo podrá existir, si existe la entidad **trabajo**, se presentan:

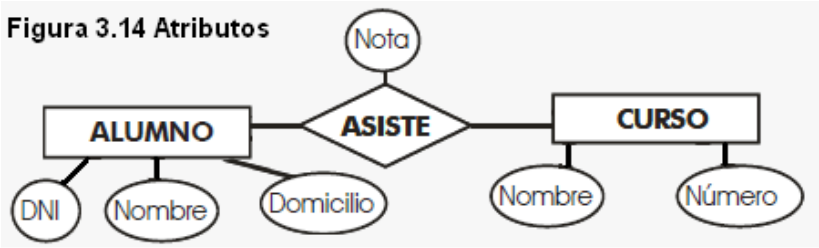
Relación: Correspondencia o asociación entre dos o más entidades, las relaciones se representan gráficamente mediante rombos y su nombre aparece en el interior.

Figura 3.13 Relación



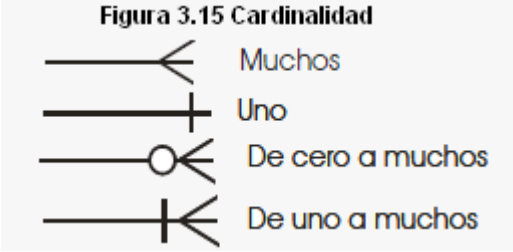
Atributo: Describen características o propiedades de las entidades y relaciones, se representan con un círculo, dentro del que se coloca el nombre del atributo, por ejemplo:

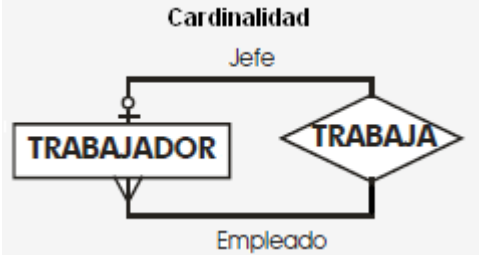
Figura 3.14 Atributos



Cardinalidad: propiedad con la que una entidad participa en una relación, específica el número mínimo y el número máximo de correspondencias en las que puede tomar parte cada ocurrencia de dicha entidad, estas pueden ser:

Figura 3.15 Cardinalidad





El diseño físico debe traducir al diseño lógico, en una solución que pueda implementarse y que sea costo-efectiva económicamente, debe cuidar el nivel de granularidad (un componente puede ser tan grande o tan pequeño según su funcionalidad, el tamaño debe proveer funcionalidad compleja y de control genérico) y agregación (un componente puede reutilizar técnicas sin duplicar código). Durante esta etapa se involucran los siguientes componentes: **(Figura 3.16)**

- **Centralización de datos:** Equivale a tener una BD maestra ubicada en un lugar central, sin copias de los datos.

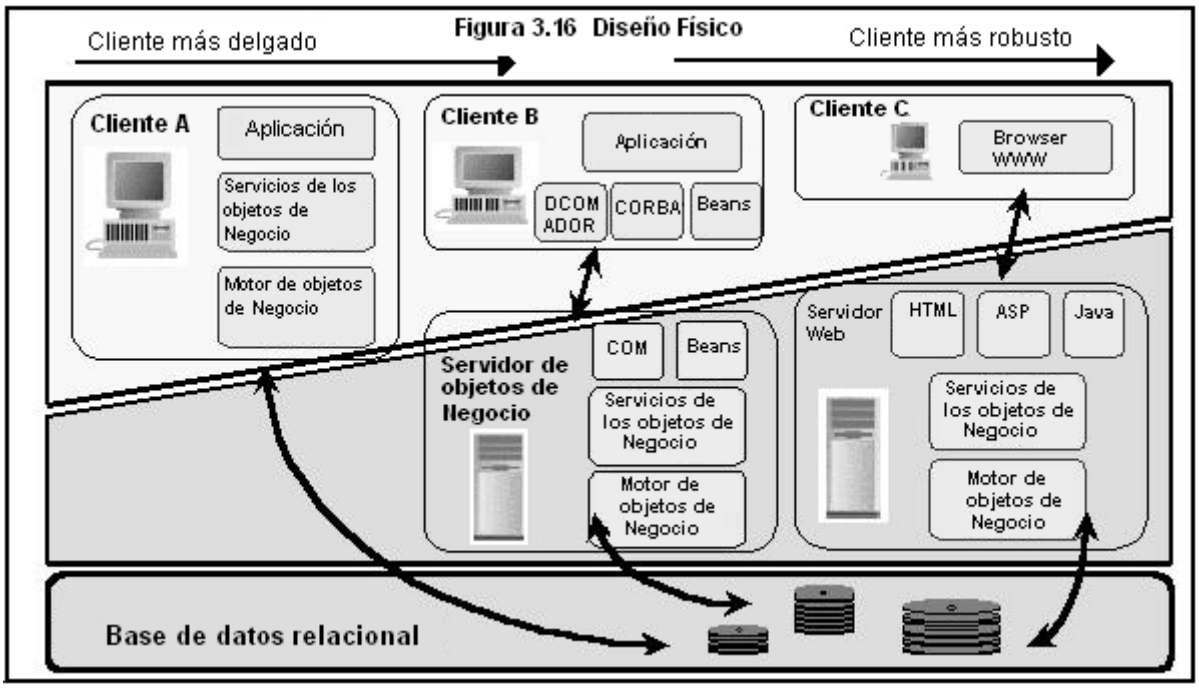
- **Partición de datos:** Se segmenta la BD maestra, resulta útil cuando se lleva a cabo fácilmente y se actualiza desde en un sitio local, no hay sobreposición entre particiones. En una partición horizontal cada hilera existe solo en una BD y en la vertical cada columna es contenida en una y solo una BD.

- **Extracto de datos:** Se realiza una copia de toda o una porción de la BD maestra, sin permitir su actualización.

- **Réplica de datos:** Fragmento de la BD maestra que puede ser actualizada, pero sin permitir actualizar la BD réplica a la vez.

El diseño físico está íntimamente ligado a una alternativa tecnológica. Ante la acelerada evolución tecnológica es importante considerar los estándares del momento y las tendencias ya que una mala decisión implicará un costo enorme (en dinero y en tiempo) al actualizarse a otra plataforma distinta.

La tendencia actual en la arquitectura cliente/servidor es crear el back-end como un servidor robusto multitareas y *multithreading* y el front-end como un cliente muy delgado que no acapare al servidor comunicándose entre sí en una plataforma internet con protocolos estándar en redes heterogéneas.



4. DESARROLLO

En esta etapa se desarrolla completamente el software y los documentos necesarios que componen el sistema. El resultado es un producto listo para que los usuarios lo puedan operar, los materiales para soporte del usuario y una descripción de la versión actual. La etapa se basa en 5 principios:

1. **Adaptar el proceso:** El desarrollo debe adaptarse a las características propias del proyecto, el tamaño del mismo, así como su tipo y regulaciones que lo condicionen, que influirán en su diseño tomando en cuenta su alcance.

2. **Balancear prioridades:** Debe encontrarse un balance que satisfaga los deseos de todos los involucrados en esta fase y principalmente al usuario.

3. **Demostrar valor iterativamente:** El proyecto debe entregarse en etapas iteradas, en cada una se analiza la opinión, la estabilidad y calidad del producto, y se refina los riesgos involucrados.

4. **Elevar el nivel de abstracción:** Este principio motiva el uso de conceptos reutilizables tales como patrón del software, lenguajes de cuarta generación (SQL, lenguajes de consulta) y tecnología (frameworks).

5. **Enfocarse en la calidad:** El control de calidad debe estar presente en todos los aspectos de la producción, ya que forma parte del proceso de desarrollo y no solo de un grupo independiente de tareas al final del desarrollo.

Se describen las actividades necesarias para la construcción, como: construir el código, planear y aplicar algunas pruebas unitarias. La construcción es un proceso iterativo, donde se refinan los componentes introduciendo elementos de lenguaje de programación, el refinamiento se repite hasta que pueda resultar el código. Para hacer las tareas de esta fase, se debe revisar los modelos creados en la fase de diseño.

4.1 Lenguaje de Programación Java

Java surge los 90`s cuando un equipo de trabajo de Sun Microsystems trato de diseñar un nuevo lenguaje de programación para electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido. Debido a la existencia de distintos tipos de arquitecturas de CPUs y a los continuos cambios, era importante tener una herramienta independiente de la arquitectura, así se desarrollo un código “neutro” que no dependía del tipo de electrodoméstico, el cual se ejecutaba sobre una “máquina hipotética o virtual” denominada Java Virtual Machine (JVM).

JVM interpretaba el código convirtiéndolo a código particular del CPU empleado, a pesar del esfuerzo, ninguna empresa de electrodomésticos se interesó en este. En 1995, ya como lenguaje de programación, se introdujo Java, se incorporo un intérprete produciendo una revolución en Internet. En 1997 apareció Java 1.1, mejorando su primer versión, en 1998 nace Java 1.2. y en 2010 Oracle lo adquiere.

Sun el creador de Java lo describe como "***simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico***".

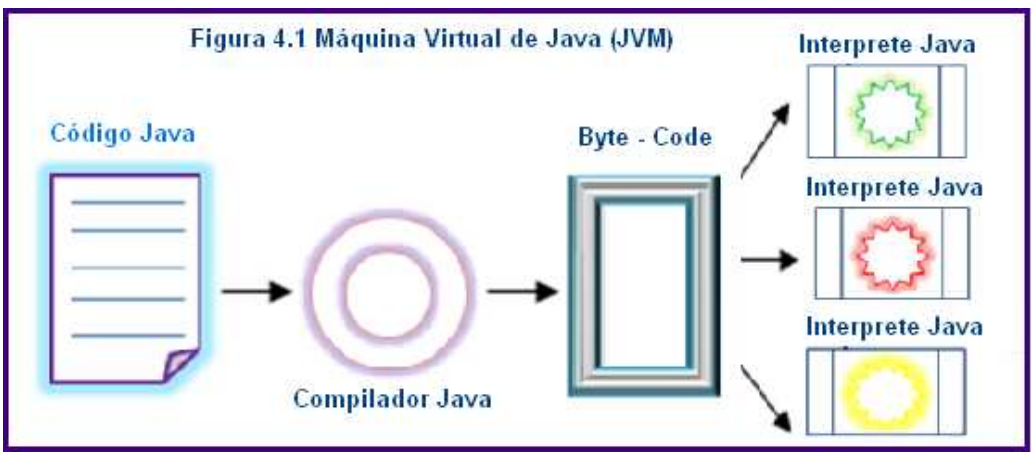
- **Herramientas de Desarrollo.**

Es necesario instalar el paquete JDK, (distribución libre) desde el sitio de Oracle, ahí mismo, se encuentra la documentación relacionada con Java (API), tutoriales, ejemplos, etc. lo que resulta de gran ayuda al momento de desarrollar aplicaciones.

- **Java Virtual Machine (JVM)**

Es una capa lógica que hace creer al programa Java que se ejecuta en una computadora, cuando en realidad sólo ve una reconstrucción lógica de éste. La existencia de distintos tipos de procesadores llevó a los creadores de Java a la conclusión de conseguir un software que no dependiera del tipo de procesador utilizado, se planteó la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina y que una vez compilado no debiera necesitar modificación por cambiar de procesador o de ejecutarlo en otra arquitectura.

JVM realiza un análisis de sintaxis del código escrito en archivos fuente (****.java***), al no encontrar errores, genera archivos compilados (****.class***), de lo contrario muestra líneas erróneas. JDK incluye el compilador **javac.exe.** y programas independientes de la plataforma.



Características del Java

1. **Sencillo y orientado a objetos:** Sencillo, no requiere gran esfuerzo de preparación para desarrollar. Orientado a objetos, ya que su enfoque es el más adecuado para las necesidades de los sistemas distribuidos y/o cliente/servidor.

2. **Robusto y seguro:** Robusto puesto que simplifica la gestión de memoria y elimina las complejidades del uso de apuntadores y aritmética de punteros del lenguaje C; seguro para poder operar en un entorno de red.

3. **Independiente de arquitectura y portable:** Java está diseñado para soportar aplicaciones instaladas en un entorno de red heterogéneo, con hardware y sistemas operativos diversos. Para que sea posible, el compilador Java genera 'bytecodes', un formato de código independiente de la plataforma diseñado para transportar código eficientemente a través de múltiples plataformas de hardware y software. El 'bytecode' es traducido a código máquina y ejecutado por la JVM.

4. **Alto rendimiento:** Dispone de diversas herramientas para su optimización, al necesitar capacidades de proceso intensivas, se usan llamadas a código nativo.

5. **Interpretado, multi-hilo y dinámico:** Incorpora capacidades avanzadas de ejecución multi-hilo (ejecución simultánea de más de un flujo de programa) y proporciona mecanismos de carga dinámica de clases en tiempo de ejecución.

4.1.1 Programación Orientada a Objetos.

La POO es una “filosofía”, un modelo de programación, con su teoría y metodología, surge como un intento para dominar la complejidad que, de forma natural posee el software. Anteriormente, se empleaba la programación estructurada para descomponer el problema en subproblemas hasta llegar a acciones simples, fáciles de codificar. La POO también descompone el problema, pero en forma de objetos, revisando lo que debe hacer el programa y su escenario real, tratando de simularlo.

Se simula en objetos, que representan el programa y contienen toda la información para abstraerlo, desde datos que lo describen, operaciones que realiza y que modelan entidades reales. Un objeto es el conjunto de variables (datos) y métodos (funciones) relacionados entre sí. Por ejemplo plancha, refrigerador y tostador, son instancias de Electrodoméstico y cuenta con atributos: marca, modelo, serie, etc; y métodos como encender, apagar. La clase tiene el propósito de categorizar objetos:

La POO se basa en objetos, que son la unidad que contiene características y comportamientos en sí misma, lo cual lo hace como un todo independiente pero que se interrelaciona con objetos de su misma clase o de otras, como sucede en el mundo real.

- **Características de la POO**

  - **Reutilización**: Capacidad de usar el mismo código en varias implementaciones; para esto se debe tener en cuenta:

  - **Polimorfismo:** Indica que un elemento puede tomar distintas formas, es decir, el uso de varios tipos en un mismo componente o función.

  - **Encapsulamiento:** Ocultamiento de información, datos o funciones especiales para los usuarios, permite que tanto la estructura (campos) como el comportamiento (métodos) se encuentren dentro del mismo cuerpo de código de la clase de creación. Dentro de la clase se agrupan datos y funciones.

  - **Herencia:** Propiedad que permite al objeto ser construidos a partir de otros; es decir recibe de un módulo superior sus características, como atributos (campos y métodos o comportamientos), heredar es compartir atributos.

  - **Sobreescritura (*override*):** Posibilidad de heredar un método y cambiarle el comportamiento en el heredero, con opción de usar el original, si se desea.

- **Clase**

Es una plantilla para agrupar objetos, es como un molde ya que a través de las clases se obtienen objetos con características entre sí. Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

- **Nombre o identificador de clase**:
  - ✓ Debe empezar con mayúscula y seguir con minúsculas (si el nombre se compone de varias palabras, la inicial de cada palabra se deja en mayúscula).
  - ✓ Sólo puede tener caracteres alfabéticos o números y empezar con mayúscula.
  - ✓ Evitar abreviaturas ayuda en la legibilidad del código. El nombre de clase debe ser claro y simbolizar perfectamente sus objetos.
  - ✓ Evitar nombres largos, se trata de que los nombres sean concisos.

- **Atributos.** Datos miembro de la clase, estos pueden ser públicos (accesibles desde otra clase) o privados (sólo accesibles por código de su propia clase.

- **Métodos.** Funciones de la clase, es lo más complicado de programar y delimitar, ya que las clases se comunican entre sí invocando a métodos.

```
[acceso] class nombreDeClase {
    [acceso] [static] tipo atributo1;
    [acceso] [static] tipo atributo2;
    ...
    [access] [static] tipo nombreMétodo1([listaDeArgumentos]) {
            ...código del método...
    }
```

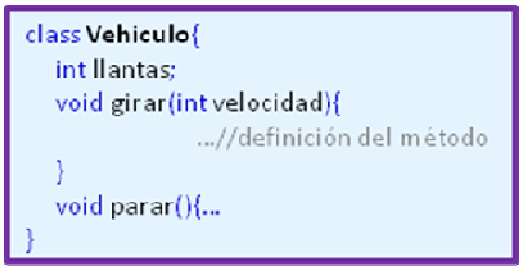
- La sintaxis en azul son las palabras reservadas (escribir exactamente así)
- La sintaxis en negro, se refiere al tipo de elemento.
- El texto entre corchetes no es obligatorio (dependiendo de las necesidades).
- Los puntos suspensivos indican que ese código se puede repetir más veces.

En Java, cada Clase debe ocupar un archivo que tiene que llamarse igual que la Clase, en donde se declara y define a la Clase. El método main (principal), no se declara en cada archivo. La mayoría de clases no tienen métodos main, sólo disponen de él las clases que se pueden ejecutar.

- **Objeto**

Se llama instancia de clase, es decir un ejemplar de la clase, se crea utilizando un constructor de clase que permite inicializar el objeto que realmente es la información que se guarda en memoria, acceder a esa información es posible gracias a una referencia al objeto.

- **Creación de la Clase:** se crea con la palabra static (opcional) para hacer que el método que lo antecede se pueda utilizar de manera genérica, los métodos y propiedades definidos así se llaman atributos y métodos de clase.



- **Creación de un objeto:** Antes de construir un objeto, se debe haber definido y creado previamente la clase a la que pertenece tal objeto. Para lo cual primero se necesita declarar una referencia al objeto, lo cual se hace igual que para declarar una variable; se indica la clase de objeto y su nombre, es decir la sintaxis es:

  Clase objeto; Ejemplo → **Vehiculo** miVehiculo;

  Esa instrucción declara que miVehiculo, hace referencia a un objeto de tipo Vehiculo, la creación del objeto se hace con el operador **new**:

  Objeto= **new Clase**(); Ejemplo → miVehiculo= **new Vehiculo()**;

- **Acceso a las propiedades del objeto:** Para acceder a los atributos de un objeto, se utiliza la siguiente sintaxis:

Objeto.atributo Ejemplo → miVehiculo.color
Para asignar valores a una propiedad: miVehiculo.color=verde;

- **Métodos:** Se utilizan de la misma forma que los atributos, solo que los métodos poseen siempre paréntesis, dentro de los cuales pueden ir valores necesarios para la ejecución del método (parámetros):

  Objeto.metodo(argumentosDelMetodo) Ejemplo → miVehiculo.gura(150);

- **Modificadores de acceso:** Palabra antepuesta a la declaración de la clase, método o propiedad, con varias posibilidades. Determina la visibilidad del elemento, por ejemplo un método que puede ser visible sólo para la clase que lo utiliza (**private**), para éstas y las heredadas (**protected**), para todas las clases del paquete (**friendly**) o para clases de cualquier tipo (**public**).

zona	private (privado)	sin modificador (*friendly*)	protected (protegido)	public (público)
Misma clase	X	X	X	X
Subclase en el mismo paquete		X	X	X
Clase (no subclase) en el mismo paquete		X		X
Subclase en otro paquete			X	X
No subclase en otro paquete				X

- **Propiedades de Clase:** Al definir los datos de una clase, se debe indicar el tipo de propiedad (String, int, double, int) y el acceso (public, private,...).

```
public class Persona {
   public String nombre;//Se puede acceder desde cualquier clase
   private int contraseña;//Sólo se puede acceder desde la
                          //clase Persona
   protected String dirección;//Acceden a esta propiedad
                              //esta clase y sus descendientes
```

- **Definición métodos:** Es una llamada a una operación de un determinado objeto, al realizar esta llamada (enviar un mensaje), el control del programa pasa a ese método y lo mantiene hasta que el método finalice. La mayoría de métodos devuelven un resultado (return), por ello hay que indicar el tipo de dato al que pertenece su resultado, si no devuelve resultado se indica como (void/vacío).

```
balón.botar(); //sin argumentos
miCoche.acelerar(10);
ficha.comer(posición15);//posición 15 es una variable que se
                        //pasa como argumento
partida.empezarPartida("18:15",colores);
```

4.1.2 Constructores.

Al crear un objeto con el operador **new**, las propiedades toman un valor inicial, ese valor puede ser el que Java otorga por defecto o el que se asignan a las propiedades, es común crear un constructor. Un constructor es un método que se invoca cuando se crea un objeto y sirve para iniciar los atributos del objeto y realizar las acciones que requiera. Siempre hay un constructor, llamado constructor por defecto, que crea el compilador si no se ha creado uno.

Por ejemplo: → Persona p= **new Persona**();

Al crear constructores se necesita un método con el mismo nombre que la clase:

```
public class Ficha {
    private int casilla;
    public Ficha() { //constructor
        casilla = 1;
    }
    public void avanzar(int n) {
        casilla += n;
    }
    public int casillaActual(){
        return casilla;
    }
}

public class App {
    public static void main(String[] args) {
        Ficha ficha1 = new Ficha();
        ficha1.avanzar(3);
        System.out.println(ficha1.casillaActual());//Da 4
    }
}
```

4.1.3 Instrucciones de comparación

Las instrucciones de un programa se ejecutan secuencialmente y este orden no altera el flujo de control del programa. Sin embargo al describir la resolución de un problema, es normal que se tenga que tomar en cuenta condiciones que influyan en la secuencia de pasos a seguir para resolverlo. Según se cumplan o no las condiciones, la secuencia involucrada será diferente, para ello las estructuras de control o comparación permiten decidir qué ejecutar y qué no en un programa.

True y false: Las instrucciones de control usan un valor verdadero o falso, para determinar su ejecución. Un ejemplo condicional es A == B, utilizando el operador condicional == para validar si A es equivalente a B, devolviendo true o false.

- **If-else:** forma básica de controlar el flujo de un programa, Else es opcional:

1. if (expresión_booleana) instrucción_si_true;
   [else instrucción_si_false;]

2. if (expresión_booleana) {
       instrucciones_si_true; }
   else { instrucciones_si_false; }

Por ejemplo:

```
public final String toString() {
        if (y<0)
                return x+"-i"+(-y);
        else
return +x+"+i"+y;
}
```

- **Switch…case…brake…default:** Ejecuta operaciones para variables con un valor entero dado, saltea todos los *case* hasta que encuentra uno con el valor de la variable, y ejecuta desde allí hasta el final del *case* o hasta que encuentre un *break*, *default* permite poner una serie de instrucciones que se ejecutan en caso de que la igualdad no se dé para ninguno de los *case*.

```
switch (expresión_entera) {
        case (valor1): instrucciones_1;
                [break;]
        case (valor2): instrucciones_2;
                [break;]
        …..
        case (valorN): instrucciones_N;
                [break;]
        default: instrucciones_por_defecto;
}
```

Por ejemplo:

```
switch (mes) {
        case (2): if (bisiesto()) dias=29;
                        else dias=31;
                        break;
        case (11): dias = 30;
                        break;
        default:    dias = 31;
        }
```

- **Operador ternario if-else:** Operador que tiene tres operandos, es un verdadero operador porque produce un valor. La expresión es la siguiente:

```
boolean-exp ? value0 : value1
```

Si *bolean-exp es* **true**, *value0* es resultado producido por el operador.
Si *boolean-exp* es **false**, *value1* es resultado producido por el operador.

Por ejemplo:

```
static int ternary(int i) {
return i < 10 ? i * 100 : i * 10;
```

4.1.4 Instrucciones para loops (ciclos)

Iteración: Los bucles de control while, do-while y for se clasifican como *instrucciones de iteración*, es decir la instrucción se repite hasta que la expresión de control booleana evaluada es falsa.

- **While**: La expresión booleana es evaluada una vez al inicio del bucle y nuevamente en cada nueva iteración de la instrucción, si la condición es falsa la primera vez, entonces la instrucción nunca se ejecuta:

```
while (expresión_booleana) {
instrucciones…
}
```

Por ejemplo:

```
while ( linea != null) {
linea = archivo.LeerLinea();
System.out.println(linea);
}
```

- **Do- While:** La diferencia entre while y do-while, es que la instrucción en do-while se ejecuta siempre por lo menos una vez, aún si la expresión se evalúa como falsa durante la primera vez. En la práctica, do-while es menos común que while.

```
do {
instrucciones…
} while (expresión_booleana);
```

Por ejemplo:

```
do {
linea = archivo.LeerLinea();
if (linea != null) System.out.println(linea);
        } while (linea != null);
```

- **For**: Un bucle for realiza un inicialización antes de la primera iteración, luego realiza pruebas condicionales y al final de cada iteración, algún incremento. La Sintaxis empleada por el ciclo for es la siguiente:

  for (inicialización; expresión booleana; paso) instrucción

  - Las tres partes del ciclo se encuentran separadas por ; (punto y coma)
  - La primer parte especifica valores previos a su inicio.
  - La segunda parte indica la condición de término del ciclo, (valor inicial)
  - La última parte especifica cómo se manipulan los valores en cada iteración.

Por ejemplo:

```
public class PruebaFor {
public static void main(String[] args)
  for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
  System.out.println("i= " + i + " j= " + j);}
    }
  }
```

4.1.5 Tipos Genéricos en Java

Tipos genéricos o parametrizados en java se introdujeron en la versión java 1.5, ayudando a resolver una gran carencia cuando se utilizan las colecciones en java.

- **Convenciones**

Por convención los tipos parametrizados son letras solitarias mayúsculas, sin esto sería difícil leer el código y decir la diferencia entre una variable parametrizada y una clase ordinaria o un nombre de una interface.

E – Elemento (Usado extensivamente en las colecciones en java)
K – Key
N – Number
T – Type
V – Value
S, U, V etc. – 2nd, 3rd, 4th types

Los Generics, permiten crear una clase que pueda operar con cualquier tipo de dato, pero este tipo no se especifica hasta que se instancie la clase. Por ello, la clase es genérica, de allí su nombre, también es llamada parametrizada por que el parámetro es especificado en su instancia, con un parámetro de tipo formal.

Aunque se puede crear clases propias que trabajen con cualquier tipo de dato de referencia (Integer, Float, Double), la mayor utilidad que se les da a los generics es en colecciones, ya que en ellas se puede definir con qué tipo de dato trabajará la misma. Por ejemplo clases ArrayList, Vector, HastTable y colecciones donde se manejaban datos Object solamente, ahora se puede especificar qué tipo de datos manejarán. Utilizando Genéricos:

1. ArrayList<String> strLista = new ArrayList<String>();
   ArrayList<Integer> intLista = new ArrayList<Integer>();

Se tiene strLista que trabajará con cadenas de caracteres y intLista con enteros, sólo agregando entre corchetes el tipo de dato. El compilador arrojará un error si en se asigna al objeto un dato que no cumpla con su norma, es decir, se puede hacer:

2. strLista.add("Cadena 1 para strLista");
   strLista.add("cadena 2 para strLista");

Algunas ventajas del uso de Generics son:

- El compilador no aceptará que se agregue ningún tipo de dato distinto al especificado en la instanciación de la clase.
- No es necesario añadir los castings que eran indispensables para recuperar los datos homogéneos de una colección Object.
- Se mantiene un mayor control sobre la colección ya que si en la versión anterior la colección hubiese aceptado el código 4, al ejecutar el código 6 hubiese ocurrido un error en tiempo de ejecución que puede ser más difícil de detectar y/o manejar.

3. Iterator strIterador = strLista.iterator();
while(strIterador.hasNext()){
String dato = (String) strIterador.next();/*Este cast ya no es necesario si se utiliza Generics*/
System.out.println(dato); }

Comodines al usar Generics: En algunos momentos se quiere que una clase genérica se comporte como una super clase de otras clases genéricas, como a continuación se muestra:

4.1.6 Wrappers

Clases que modelan los tipos de datos primitivos como enteros y flotantes, que son los únicos elementos en Java que no son clases. Una diferencia entre los primitivos y sus wrapper: Los primitivos se pasan como argumento a los métodos por valor, mientras que los objetos se pasan por referencia. que implica ventaja de eficiencia. Estas clases se encuentran en java.lang, derivan de Number que deriva de Object. Cada una de estas clases contiene un valor primitivo del tipo relacionado, por ejemplo un objeto Integer contiene un int como atributo.

Métodos de Integer: El constructor de un Wrapper está sobrecargado para aceptar el tipo de dato primitivo que va a contener o un objeto String.

Integer(int)
Integer(String)

Existen métodos que permiten recuperar el tipo primitivo que se quiera.

doubleValue()
floatValue()

También tenemos convertidores con la clase String.

String toString()
Integer valueOf(String)

Estos métodos se utilizan en forma estática, es decir, no es necesario crear una instancia para utilizarlos. Aunque no es del todo cierto, ya que como sucede con el método valueOf(), retorna una instancia del wrapper. Por lo que las clases wrapper son fabricantes de objetos.

Conversiones entre distintos tipos de datos básicos

El procedimiento para la conversión en cada uno de los métodos es el mismo: crear una instancia del wrapper del argumento, con el valor de dicho argumento, y luego solicitar el tipo de dato primitivo a retornar por el método de conversión.

```
final class Cast
{
/*float a int*/
public static int toInt(float f)
{    int I;
i= Float.valueOf(f).intValue();
return I;   }  }
```

Por ejemplo:

```
int i;
float f;
doublé d;
String s=”9.738”;
d= Cast.toDouble(s);  // d=9.738
Cast.toInt(d); //i=9
f= I; // f= 9.0
s= Cast.toString(f); // s= ”9.0”
```

4.2 Lenguaje de Programación Avanzada.

4.2.1 Clase Date

En todos los programas java, viene importada por defecto la librería java.lang, y dentro de ella sus clases disponibles, java.lang no posee a Date, por lo que se debe importar para poder usarla. Date se encuentra en la librería estándar java.util, para traer cualquier clase de esta u otras librerías, al inicio del programa se debe colocar la palabra import y el nombre de la librería, si solo se requiere a Date se coloca:

```
import java.util.Date;
```

Aunque lo más recomendable es incluir todas las Clases de la librería, por si es requerida alguna adicional, mediante un asterisco como a continuación se muestra:

```
import java.util.*;
```

Ejemplo, desplegar en pantalla la fecha actual, para lo cual se debe utilizar un println(), que le dice a la maquina “imprime lo que te estoy dando a la consola y termina con salto de línea”. Por lo que se escribirá System.out.println(“cadena de caracteres”) y debe contener un método principal main() como se muestra:

public static void main(String[] args) {

La palabra clave public significa que el método está disponible para el mundo exterior. El argumento para el método main() es un arreglo de objetos String. Los args no se utilizan en este ejemplo. La línea que imprime la fecha es la siguiente:

System.out.println(new Date());

Por ejemplo:

```
// HolaFecha.java
import java.util.*;                    //importando librería útil que contiene la
Clase Date
public class HolaFecha {               //nombre de la Clase que es pública
public static void main(String[] args) {   //método principal
System.out.println("Hola, hoy es: ");      //desplegando en pantalla la cadena
indicada
System.out.println(new Date());        //desplegando en pantalla la fecha actual
}          }
```

4.2.2 Clase Calendar

Java posee otras clases, que en conjunto nos facilitan realizar operaciones como suma, resta, obtener un día y hora exactos, entre otras. Estas funcionalidades se encuentran en dos clases; Calendar que permite obtener campos enteros como día, mes y año de objetos de tipo java.util.Date o que hereden de él. Y GregorianCalendar implementación del calendario gregoriano.

- **Calendar:** Clase abstracta para convertir objetos de tipo Date (java.util.Date) con un conjunto de campos como YEAR (año), MONTH (mes), DAY (día), HOUR (hora), etc.

Calendar tiene comportamiento de java.util.Date, es decir, al obtener una instancia de Calendar se obtiene un instante de tiempo específico con gran precisión similar a lo obtenido con Date, que tiene métodos para obtener año, mes y día, pero que son obsoletos, ya que por eso existe Calendar. Y cuando se usa el método getYear() de java.util.Date esta recurre a las funcionalidades que posee Calendar.

Obtener el tiempo específico o diferente al actual es fácil, solo se indica el día, mes y año con a trabajar, o especificado la hora, minuto y segundo. Por ejemplo: getInstance()devuelve una subclase Calendar con el tiempo ajustado a la hora actual, y usando el método set(args…) se fuerza a tomar la fecha deseada:

```
Calendar ahoraCal = Calendar.getInstance();
System.out.println(ahoraCal.getClass());
ahoraCal.set(2010,10,7);
System.out.println(ahoraCal.getTime());
ahoraCal.set(2010,10,7,7,0,0);
System.out.println(ahoraCal.getTime());
```

La primera línea trae una instancia de GregorianCalendar con la fecha y hora actual; el método getTime() retorna un objeto de tipo java.util.Date que se muestra en pantalla. En la tercera línea se usa el método set para ajustar la fecha al "7 de octubre de 2010" y en la quinta línea se pone la misma fecha ajustando hora, minuto y segundo:

```
Thu Oct 07 19:55:47 GMT-05:00 2010
Thu Oct 07 07:00:00 GMT-05:00 2010
```

Extraer datos: Calendar tiene un único método get para obtener datos y se ayuda de atributos que le permiten obtener o ajustar la fecha, como son:

- YEAR: Año, MONTH: Mes.
- DATE, DAY_OF_MONTH: Día del mes.
- DAY_OF_WEEK: Día de la semana entre 1 (MONDAY) y 7 (SATURDAY).
- HOUR: Hora antes o después del medio día (en intervalos de 12 horas).
- HOUR_OF_DAY: Lo hora absoluta del día (en intervalos de 24 horas).
- MINUTE: El minuto dentro de la hora.
- SECOND: El segundo dentro del minuto.

Tiene atributos en meses: january, march, december, etc. desde 0 (january) hasta 11 (december); y atributos en días: sunday, tuesday, etc. 1 (sun) 7 (sat).

Por ejemplo:

```
System.out.println("ANYO: "+ahoraCal.get(Calendar.YEAR));
System.out.println("MES: "+ahoraCal.get(Calendar.MONTH));
System.out.println("DIA: "+ahoraCal.get(Calendar.DATE));
System.out.println("HORA: "+ahoraCal.get(Calendar.HOUR));
if (ahoraCal.get(Calendar.MONTH) == Calendar.OCTOBER){
 System.out.println("ES OCTUBRE");
}else{
 System.out.println("NO ES OCTUBRE");
}
```

Si el objeto tiene fecha equivalente a 5:30 p.m. de 10 de octubre de 2010 se obtiene:

```
ANYO: 2010
MES: 9
DIA: 10
HORA: 5
ES OCTUBRE
```

4.2.3 Clase File

Define métodos para conocer propiedades del archivo (última modificación, permisos de acceso, tamaño...); y métodos para modificar alguna característica del archivo. Los constructores de esta clase permiten inicializar el objeto con el nombre de un archivo y la ruta donde se encuentra o inicializar el objeto con otra ruta y nombre distintos.

- Crea un File, con el nombre y ruta del archivo pasado como argumento.
  public File(String nombreCompleto)

- Crea un File, con primer argumento ruta y el segundo con nombre de archivo.
  public File(File ruta, String nombre)

- Devuelve true si el archivo existe (o el directorio)
  public boolean exists()

- Devuelve true si puede escribir en el archivo, si no es de sólo lectura.
  public boolean canWrite()

- Devuelve el número de bytes del archivo. Si es directorio devuelve cero.
  public long length()

- Devuelve la hora de última modificación (Hr, Min, Seg)
  public long lastModified()

Por ejemplo:

```
…..
File miF;
String cd;
System.out.println("Se teclea un archivo con su ruta o directorio");
        do  {
          cd=Leer.dato();
          miF= new File(cd);
          if(miF.exists())     {
              System.out.print(cd="esta en el disco ");
              if(miF.isFile())
        System.out.println("Es un archivo y tiene como tamaño: "+miF.length());  else
        if(miF.isDirectory())
        System.out.println("Es un directorio ");
        else
        System.out.println("Se desconoce que es ");  }
         else
         if(cd.length() > 0)
        System.out.println("No existe el elemento en disco");
        }while(cd.length()>0);      }   }
```

4.2.4 Clase NumberFormat

Una de las operaciones más comunes a realizar, es trabajar con números dándoles formato, ya sea con el fin de cumplir con los requerimientos de impresión o

simplemente mostrar los datos de una manera más agradable y legible al usuario. Java proporciona la clase NumberFormat (java.text.NumberFormat) para realizar este tipo de operaciones; los principales métodos, de esta clase son:

setMinimumIntegetDigits
setMinimumFractionDigits
setMaximumIntegerDigits
setMaximunFractionDigits.

Por Ejemplo:

```
import java.text.NumberFormat;
public class formato  {
public static void main (String[] argv){
double value = 361.23456789; // 8 digitos en la parte fraccionaria y 3 dígitos en la
parte entera.
NumberFormat nf = NumberFormat.getInstance();     // obtener el formato  numérico
actual.
nf.setMaximumFractionDigits(5); //redondea a 5 dígitos 361.23457
System.out.println(nf.format(value));
nf.setMaximumFractionDigits(10); //mostrar mas digitos fraccionales de los que tiene
361.23456789
System.out.println(nf.format(value));
nf.setMaximumIntegerDigits(2); //definir menos enteros de los que tiene 61.2345678
System.out.println(nf.format(value));
 nf.setMaximumIntegerDigits(5); //definir mas digitos enteros de los que tiene
61.23456789
System.out.println(nf.format(value));
 } }
```

4.2.5 Clase DateFormat

Los objetos Date, como ya se vio representan fechas y horas, pero no se puede mostrar o imprimir objetos Date sin ser convertidos primero a una cadena (String). El formato de estos objetos debería ir conforme a las convenciones de la Localidad donde se esté usando, puesto que, para los alemanes será adecuado usar 9.4.98 como una fecha válida, pero para los norteamericanos la misma fecha aparecería como 4/9/98.

Formatos Predefinidos

DateFormat proporciona estilos de formateo predefinidos específicos de cada Localidad, aunque no soporta todas las posibles definiciones, para ver las definiciones de Locale que reconoce DateFormat, se debe llamar al método getAvailableLocales.

```
Locale[] locales = DateFormat.getAvailableLocales();
```

Fechas

Formatear fechas con DateFormat es un proceso sencillo, se crea un método getDateInstance para formatear; después, se llama a format, que devuelve un String y contiene la fecha formateada. Para obtener la fecha actual:

```
Date today;
String dateOut;
DateFormat dateFormatter;
dateFormatter = DateFormat.getDateInstance(DateFormat.DEFAULT, currentLocale);
today = new Date();
dateOut = dateFormatter.format(today);
System.out.println(dateOut + "   " + currentLocale.toString());
```

DEFAULT, es un formato predefinido de los que proporciona DateFormat, otros son:

```
DEFAULT
SHORT
MEDIUM
LONG
FULL
```

La siguiente tabla muestra el formateo de cada estilo para ciertas localidades:

Estilo	Localidad U.S.	Localidad Francia
DEFAULT	10-Apr-2010	10 avr 10
SHORT	4/10/10	10/04/10
MEDIUM	10-Apr-10	10 avr 10
LONG	April 10, 2010	10 avril 2010
FULL	Friday, April 10, 2010	vendredi, 10 avril 2010

Horas: Para formatear horas con DateFormat se hace de manera similar al formateo de fechas, excepto en que el tipo de formateo se crea con el método getTimeInstance.

```
DateFormat timeFormatter =
DateFormat.getTimeInstance(DateFormat.DEFAULT, currentLocale);
```

En la siguiente tabla se observan estilos de formatos de hora en distintas localidades.

Estilo	Localidad U.S.	Localidad Alemania
DEFAULT	3:58:45 PM	15:58:45
SHORT	3:58 PM	15:58
MEDIUM	3:58:45 PM	15:58:45
LONG	3:58:45 PM PDT	15:58:45 GMT+02:00
FULL	3:58:45 oclock PM PDT	15.58 Uhr GMT+02:00

Fechas y Horas

Para mostrar fecha y hora en un String, se crea un formateador con el método getDateTimeInstance, donde el primer parámetro es el estilo de fecha y el segundo es el estilo de la hora; el tercer parámetro Locale. Por ejemplo:

```
DateFormat formatter;
formatter = DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG,currentLocale);
```

Formatear con Patrones

Con SimpleDateFormat, se crean formatos personalizados específicos de la Localidad, ya que se específica un patrón String, sus contenidos determinan el formato de la fecha y de la hora.

```
Date today;
String output;
SimpleDateFormat formatter;
formatter = new SimpleDateFormat(pattern, currentLocale);
today = new Date();
output = formatter.format(today);
System.out.println(pattern + "    " + output);
```

Patrón	Salida
dd.MM.yy	09.04.10
yyyy.MM.dd G 'at' hh:mm:ss z	2010.04.09 AD at 06:15:55 PDT
EEE, MMM d, "yy	Thu, Apr 9, '10
h:mm a	6:15 PM
H:mm	18:15
H:mm:ss:SSS	18:15:55:624
K:mm a,z	6:15 PM,PDT
yyyy.MMMMM.dd GGG hh:mm aaa	2010.October.09 AD 06:15 PM

Patrones y Localidades

SimpleDateFormat es sensible a la localidad. Si se ejemplifica SimpleDateFormat sin el parámetro Locale, formateará la fecha y hora de acuerdo a la Localidad por defecto. Tanto el patrón como la Localidad determinan el formato. Para el mismo patrón, la claseSimpleDateFormat puede formatear la fecha y la hora de forma diferente si varía la Localidad.

```
Date today;
String result;
SimpleDateFormat formatter;
formatter = new SimpleDateFormat("EEE d MMM yy", currentLocale);
today = new Date();
result = formatter.format(today);
System.out.println("Locale: " + currentLocale.toString());
System.out.println("Result: " + result);
```

4.2.6 Interface List y clases hijas

Un tema complicado de organizar es una colección de objetos, existe una estructura de datos; list que almacena un solo objeto y su tiempo de acceso es mínimo; otra es el arreglo *array*, que permite obtener el valor de una posición en una tabla, es rápido y su equivalencia se organiza en la memoria de la computadora. Los valores en un arreglo están en orden, y en ese mismo orden están dispuestos en la memoria, por lo tanto el acceso lo maneja directamente el hardware (la memoria funciona físicamente como un arreglo).Un arreglo en Java se declara como:

```
String[] nombres = new String[10];
```

Se pueden usar arreglos para todo, si se requiere guardar las calificaciones de alumnos, el usar un arreglo puede ser: se crea una clase con el nombre del alumno y su calificación:

```
class Nota {
  String alumno;
  int nota;
Nota(String alumno, int nota) { this.alumno = alumno; this.nota = nota; }
  public int getNota() { return nota; }
  public String getAlumno() { return alumno; }   }
```

Para almacenar las calificaciones de 500 alumnos:

```
Nota[] notas = new Nota[500];
notas[0] = new Nota("Juan", 7);
notas[1] = new Nota("Pedro", 8);
// …
notas[499] = new Nota("Elías", 9);
```

¿Qué sucede si se requiere saber la calificación de algún alumno? No se sabría en qué posición del arreglo esta, la única manera de saberlo es recorrer el arreglo:

```
for(Nota n : notas)
        if(n.getNombre().equals("Elías"))
                return n.getNota();
```

Esto consume tiempo, ya que encontraría después de efectuar 499 comprobaciones, mediante algoritmos, se reduce la necesidad de comprobar tantas veces. Se debe tener en cuenta que se necesita mantener un arreglo ordenado, pero, ¿qué pasa si se agrega un alumno? se inserta en la posición correcta de acuerdo al orden alfabético y desplazar todos alumnos a su derecha. El tipo de estructura depende de lo que puede asumir la colección almacenada y de cómo se usa la colección.

- **Colecciones (Collection)**

Collection es todo aquello que se puede recorrer (o “iterar”) y de lo que se puede saber el tamaño, existen muchas clases que extienden de Collection imponiendo más restricciones y dando más funcionalidades.

Cabe resaltar que el requisito de que se sepa el tamaño, hace inconveniente utilizar estas clases con colecciones de objetos de las que no se sepa la cantidad (lo que podría considerarse una limitante).

add(T): Añade un elemento.
iterator(): Obtiene un iterador, recorriendo la colección un elemento cada vez.
size(): Obtiene la cantidad de elementos que esta colección almacena.
contains(t): Pregunta si el elemento t ya está dentro de la colección.

Hay cuatro interfaces que extienden Collection: List, Set, Map y Queue. Cada una de agrega condiciones sobre los datos que almacena y como contrapartida ofrece más funcionalidad.

i. List (Lista)

List es un Collection, la lista mantiene un orden arbitrario de elementos y permite acceder a ellos por orden. No hay ningún método en Collection para obtener el tercer elemento, pues no existe seguridad de saber que al recorrer una colección por segunda vez los elementos aparezcan en el mismo orden que la primera y al contrario una lista sí lo permite:

get(int i): Obtiene el elemento en la posición i.
set(int i, T t): Pone al elemento t en la posición i.

Existen dos implementaciones de List, una **ArrayList**, su ventaja es que es expansible, es decir, crece a medida que se añaden elementos (array es fijo desde su creación). El tiempo de acceso a un elemento es mínimo y lo malo es que al eliminar un elemento del principio, o de en medio, la clase debe mover todos los elementos que le siguen a la posición anterior, para tapar el agujero del elemento removido; lo que resulta costoso.

Otra es **LinkedList** (lista enlazada), donde los elementos se mantienen en nodos atados entre sí cada nodo apunta a su antecesor y al elemento que le sigue. No hay nada en los nodos que tenga algo que ver con la posición en la lista, así para obtener el elemento “n”, se necesita empezar desde el primer nodo, e ir avanzando al siguiente n veces; para buscar el elemento 400, se requerirían 400 de pasos. Su ventaja es que se eliminan elementos del principio o de en medio de manera eficiente, solo se modifica a sus vecinos, y s e ignora al elemento borrando.

get(int) es lento porque, necesita recorrer todo para llegar al elemento pedido, esto hace que recorrer la lista con for(int i = 0 ; i < lista.size(); i++) sea lento. LinkedList sólo debe recorrer mediante iteradores.

ii. Set (Conjunto)

Set es un Collection y agrega una sola restricción: No puede haber duplicados. Es posible que existan sets que aseguren un orden determinado al recorrerlo (obtener strings en orden alfabético), ese orden no es arbitrario, ya que la interfaz Set no tienen ninguna funcionalidad para manipularlo (como List). Una ventaja es que pregunta si un elemento ya está contenido mediante contains lo que es conveniente utilizar cuando no importe el orden y se necesite preguntar si un elemento está o no.

Existen varias implementaciones de Set, la más común es HashSet. Sets y Maps aprovechan cierta característica de Java: Todos los objetos heredan de Object, por lo tanto todos los métodos de la clase Object están presentes en todos los objetos. Dos de estas características son:

- Saber si es igual a otro, con su método equals().
- Devolver un número entero de modo, que si dos objetos son iguales ese número también lo será (hash), método hashCode().

HashSet: aprovecha la segunda característica, a cada objeto que se añade se le pide calcular su hash (valor entre -2147483647 y 2147483648), ese valor se guarda en una tabla y más tarde, cuando se pregunta con contains() si un objeto x ya está, habrá que saber si está en dicha tabla. Pero ¿En qué posición está? HashSet puede saberlo, ya que para un objeto determinado, el hash siempre va a tener el mismo valor, la función contains de HashSet saca el hash del objeto y va con eso a la tabla. En la posición de la tabla hay una lista de objetos que tienen ese valor de hash, y si uno de esos es el buscado contains devuelve true.

El orden en el que aparecen los objetos al recorrer Set es impredecible, es importante notar que la función hashCode() tiene que devolver siempre el mismo valor para objetos iguales (equals() da true), ya que si no es así, HashSet pondrá al objeto en una posición distinta en la tabla y al consultarla con contains da falso, por más sea correcto el add. Lo mismo sucede si se usan claves de objetos que varíen.

TreeSet: Una cosa que pueden saber los objetos con independencia de cómo y dónde son usados es saber ordenarse, a diferencia de equals y hashCode, que están en todos los objetos, la capacidad de ordenarse, se presenta sólo en aquellos que implementan la interfaz Comparable. Cuando un objeto implementa esta interfaz promete saber compararse con otros (método compareTo()), y responder si él está antes, después o es igual al objeto que se le pasa como parámetro. Al orden de usar este método se le llama orden natural, algunas clases en implementan Comparable, como String, y su orden natural es alfabético, e implementan Date, Number, etc.

Si se crea una clase llamada Alumno, la definición de un orden natural para esta puede ser el apellido, el nombre, el número de matrícula, etc., de acuerdo al atributo que se elija para definir el orden natural, se codifica el método compareTo().

iii. Map

Map representa un diccionario y suele asociarse a la idea de tabla hash, Map es un conjunto de valores, con el detalle de asociarles un objeto extra. A los primeros se les llama claves o keys, ya que permiten acceder a los segundos. Al decir que las claves forman un conjunto, se enfoca a que se ven como un Set, ya que no puede haber duplicados, la única manera de lograr esto es haciendo que Map guarde listas de valores en vez de los valores sueltos.

Map no es Collection, ya que la interfaz le queda chica, se puede decir que Collection es unidimensional, mientras que Map es bidimensional y no hay una manera fácil de expresar un Map en una serie de objetos que podemos recorrer, pero sí podríamos recorrer una serie de objetos si cada uno representa un par {clave, valor}, aunque esta forma de recorrer Map no es la principal forma en que se usa. Algunos métodos importantes de Map son:

- get(Object clave): Obtiene el valor correspondiente a una clave, devuelve null si no existe esa clave en el map.
- put(K clave, V valor): Añade un par clave-valor a map, si ya existía uno lo reemplaza.

iv. Queue y Deque (Cola)

Es una colección diseñada para ser usada como almacenamiento temporal de objetos a procesar, las operaciones que suelen admitir las colas son encolar, obtener siguiente, etc. Por lo general siguen un patrón FIFO (First In - First Out - Lo que entra primero, sale primero), el orden en que se van obteniendo los siguientes objetos coincide con el orden en que fueron introducidos en la cola. Lo análogo sucede en el supermercado: la gente que hace la cola va siendo atendida en el orden en que llegó a ésta. La interfaz Queue tiene las operaciones que se esperan en una cola y Deque, que representa a una “double-ended queue”, es decir, una cola donde los elementos pueden añadirse al final y empujarse al principio.

4.2.7 Java SWING

AWT (Abstract Windows Toolkit) se ocupa de construir interfaces gráficas de usuario y se complementa con el paquete Swing. Los componentes AWT se llaman componentes pesados debido al uso que hacen de los recursos del sistema y los Swing que son componentes ligeros, ya que no necesitan ventanas propias para ejecutarse; para cada componente Swing hay un AWT (ventanas, marcos, applets y

diálogos utilizan Swing para visualizar componentes basados en contenedores AWT). A continuación una lista con las clases de AWT más populares:

- Applet: Crea un applet.
- Button: Crea un botón.
- Canvas: Crea un área de trabajo en el que se puede dibujar.
- Checkbox: Crea una casilla de activación.
- Menu: Crea un menú.
- ComboBox: Crea un cuadro de lista desplegable.
- List: Crea un cuadro de lista.
- Frame: Crea un marco para las ventanas de aplicación.
- Panel: Crea un área de trabajo que puede tener otros controles.
- RadioButton: Crea un botón de opción.
- TextArea: Crea un área de texto de dos dimensiones.
- TextField: Crea un cuadro de texto de una dimensión.

Applet

Programa ejecutado dentro de un navegador Web, los applets se incluyen en páginas Web utilizando etiquetas HTML <APPLET>, al ejecutarlos se descargan automáticamente, el browser los despliega, y se visualizan en la página. Pueden hacer de todo, trabajar con gráficos, visualizar animaciones, gestionar cuadros de texto y botones, haciendo páginas Web activas. Sus características son:

- No tienen método main() para ejecutarse, este papel lo toman otros métodos.
- Todos los applets derivan de la clase java.applet.Applet.
- Redefine métodos heredados de Applet al ejecutarse: init, start, stop, destroy.
- Interfaces gráficas heredadas de Component, Container y Panel.

Se construye un marco de trabajo de JApplet y se sobrescriben los métodos como:

Método	Operación
init()	Automáticamente llamado para realizar la inicialización del inicio del applet, incluyendo el diseño de los componentes. Siempre se sobrescribirá este método.
start()	Llamado cada vez que el applet se mueve dentro de la vista del navegador Web para permitir que el applet comience su operación normal (especialmente aquellos que tienen que apagarse con **stop()**). También llamada luego de **init()**.
stop()	Llamado cada vez que el applet se mueve fuera de la vista del navegador Web para permitir que el applet apague operaciones de alto consumo. También llamado inmediatamente antes de **destroy()**.
destroy()	Llamado cuando el applet esta siendo descargado de la página para realizar la liberación final de recursos cuando el applet no se va a utilizar mas.

Además existen los métodos paint y update que:

- **Paint:** Es llamado cuando se vuelve a dibujar un applet.
- **Update:** Es llamado cuando solo se vuelve a dibujar una parte del applet. Se rellena el applet con otro color de fondo antes de volver a dibujarla.

Se crea un applet, basado en java.applet.Applet que se basa en Component AWT. A continuación un ejemplo:

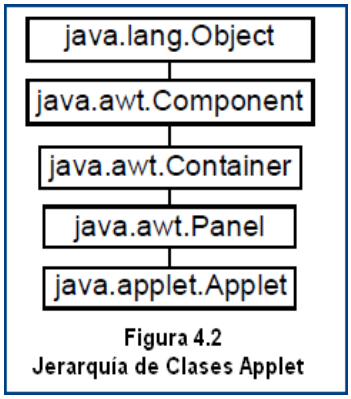


Figura 4.2
Jerarquía de Clases Applet

Por ejemplo:

```
import java.applet.App1et;
import j ava . awt . * ;
public class applet extends Applet{
public void paint(Graphics g){
g.drawString("iHola desde Java!", 60, 100);}
}
```

- **Aplicaciones**

Las ventanas AWT están basadas en la clase Frame, que crea una ventana con marco para visualizar botones y títulos, permitiéndole al usuario interactuar con el programa por medio de eventos. Cuando el usuario ejecuta alguna acción, (dar clic en un botón, cerrar una ventana, seleccionar un elemento o usar el ratón), Java lo considera un evento.

La clase básica de AWT es java.awt.Component y es donde se basan todos los componentes visuales de AWT, como la clase Button, la clase java.awt. Component, la clase Container, etc. Sus métodos se pueden sobrescribir para personalizarlos como sea, al sobrescribir paint se puede dibujar una cadena de texto; al sobrescribir init se cambia el color de fondo del applet. Al construir una interface grafica se hace:

1. Un contenedor (container): Ventana donde se sitúan los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos.
2. Los componentes: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc.
3. Modelo de eventos: el usuario controla la aplicación actuando sobre los componentes, con el ratón o con el teclado. Cada vez que se realiza una acción, se produce el evento correspondiente, que el S.O. transmite al AWT.

AWT crea un objeto que determinada el tipo de evento, derivado de Event, tal evento es transmitido a un método para su gestión. El componente u objeto que recibe el evento debe “registrar” qué objeto se va a hacer cargo de gestionar tal evento.

- **Gráficas Primitivas**

Java dispone de métodos para realizar dibujos sencillos, llamados gráficos primitivos, donde las coordenadas se miden en pixels, empezando a contar desde cero. Excepto los polígonos y las líneas, todas las formas geométricas se determinan por el rectángulo que las comprende, cuyas dimensiones son w y h, los polígonos admiten un argumento de la clase java.awt.Polygon.

- **Imágenes**

Java permite incorporar imágenes de tipo GIF y JPEG definidas en archivos por medio de java.awt.Image, para cargar una imagen se indica la localización del archivo (URL) y cargarlo con el método Image getImage(String) o Image getImage(URL, String). Tales métodos pertenecer a java.awt.Toolkit y java.applet.Applet. Para cargar una imagen hay que comenzar creando un objeto Image y llamar al método getImage(), pasando como argumento la URL.

```
Image miImagen = getImage(getCodeBase(), "imagen.gif")
```

Una vez cargada la imagen, hay que representarla mediante el método paint() y llamar al método drawImage() de la clase Graphics, este método admite varias formas incluyendo el nombre del objeto imagen, las dimensiones y ImageObserver.

4.3 Java Conectividad de la Empresa.

Los componentes de software en aplicaciones distribuidas que interactúan en empresas, se pueden visualizar capas lógicas, las cuales representan la independencia física y lógica de los componentes de software en función de la naturaleza de los servicios que proporcionan. La dimensión de capas lógicas de la arquitectura se muestra de la siguiente forma **(figura 4.3)**:

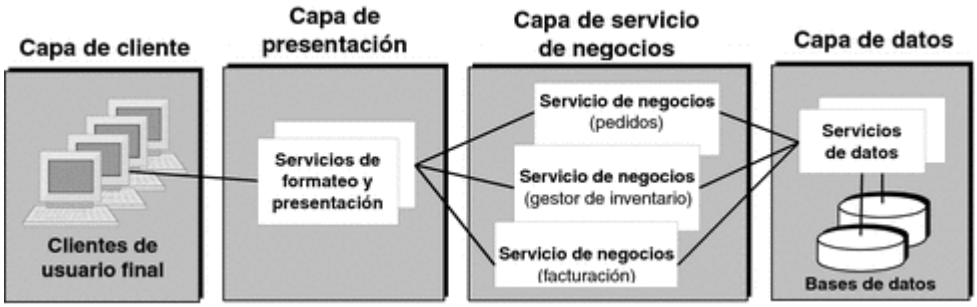


Figura 4.3 Dimensión en 2 capas lógicas para aplicaciones distribuidas

- ✓ **Capas lógicas:**

Capa de cliente: Formada por la lógica de la aplicación a la que el usuario final accede mediante una interfaz de usuario, incluye clientes basados en navegadores, componentes Java ejecutados en equipos de escritorio o clientes.

Capa de presentación: Formada por la lógica de aplicación, que prepara datos para su envío a la capa de cliente y procesa solicitudes desde la capa de cliente para su envío a la lógica de negocios del servidor. La lógica en la capa de presentación se forma por Servlet o JSP que preparan datos para enviarlos en formato HTML o XML.

Capa de Negocios: Es la lógica que realiza las funciones principales de la aplicación: procesamiento de datos, implementación de funciones de negocios, coordinación de usuarios y administración de recursos externos como, bases de datos o sistemas heredados. Suele estar formada por componentes acoplados que se ajustan al modelo de componentes distribuidos J2EE como EJB o beans.

Capa de datos: Formada por los servicios que proporcionan los datos persistentes utilizados por la lógica de negocios. Los datos pueden ser datos de aplicaciones almacenados en un sistema de administración de bases de datos o pueden incluir información de recursos y directorios almacenada en un almacén de datos de protocolo ligero de acceso a directorios (LDAP).

- ✓ **Independencia lógica y física**

Independencia lógica. Las cuatro capas representan independencia lógica, ya que se puede modificar la lógica de la aplicación de una capa independientemente de la lógica de las otras. Puede cambiar la implementación de la lógica de negocios sin tener que cambiar o actualizar la lógica de la capa de presentación o la de cliente.

Independencia física. Es posible implementar la lógica en capas distintas para varias plataformas de hardware (configuraciones de procesador, conjuntos de chips y sistemas operativos). Que permite ejecutar componentes distribuidos en equipos que mejor se adapten a necesidades individuales y maximizar recursos.

La forma de asignar componentes de aplicación o componentes de infraestructura al hardware depende de varios factores, en función de la escala y la complejidad de la solución de software, ya que en implementaciones a gran escala, la asignación de componentes toma en cuenta la velocidad y potencia de equipos, ancho de banda, consideraciones de seguridad, estrategias de escalabilidad y alta disponibilidad.

4.4 Java Programación para Desarrolladores.

Internet hoy en día es la red de computadoras más extensa del planeta, enlaza miles de millones de redes locales heterogéneas, basándose en páginas hipertexto (www), que no es nuevo (introducido en 1965), y se define como texto de recorrido no secuencial, en el que dando clic a enlaces (links) se accede al documento que apunta y que normalmente contiene información representada por el enlace.

No solo existe Internet, sino también Intranet (basada en hipertexto pero con ámbito limitado) que se reducen al marco de una empresa, institución o centro educativo y carecen de interés para usuarios externos, debido al tipo de información que ofrecen y la mayoría de los casos su acceso está prohibido o restringido a usuarios externos.

- **Clientes (clients)**

Por su versatilidad y potencialidad, actualmente la mayoría de usuarios de Internet utilizan servidores de datos mediante navegadores, cada uno cumple con la mayoría de los estándares aceptados en Internet, y también proporcionan soluciones adicionales a problemas específicos, por lo cual es necesario tener en cuenta qué el browser se comunica con el servidor, y el resultado varia dependiendo del browser.

Todos los browsers soportan Java, lo cual implica que disponen de una JVM donde se ejecutan los archivos *.class de los Applets, además de que tienen la posibilidad de sustituir la JVM por medio de un mecanismo definido por Sun, que se llama Java Plug-in (aplicaciones que se ejecutan controladas por los browsers y permiten extender capacidades, para soportar nuevos formatos de audio o video).

- **Servidores (servers)**

Programas que se encuentran esperando a que alguna computadora realice una solicitud de conexión, es posible que en el mismo servidor se tenga simultáneamente servidores con distintos servicios (HTTP, FTP, TELNET, etc.). Cuando al servidor le llega un requerimiento de servicio enviado por otro servidor o computadora a través de la red, se interpreta la llamada, y se pasa el control de conexión al servidor correspondiente. Si no se tiene el servidor adecuado para responder, está es rechazada **(figura 4.4)**.

No todos los servicios actúan igual, algunos, como FTP, una vez establecida la conexión, la mantienen hasta que el cliente o el servidor explícitamente la cortan. Por ejemplo, al establecer una conexión FTP, los extremos se mantienen en contacto hasta que el cliente cierre la conexión (quit, exit) o pase un tiempo sin actividad entre ambos.

La mayoría de usuarios de Internet son clientes que acceden mediante un browser a los servidores WWW, el servidor no permite acceder indiscriminadamente a todos sus archivos, sino únicamente a determinados directorios y documentos previamente establecidos por el administrador. Actualmente, la mayoría de aplicaciones de entornos empresariales están construidas en arquitecturas cliente-servidor. Los clientes son PCs de menor capacidad de procesamiento y orientados a usuario final; los servidores son intermediarios entre clientes y servidores especializados (grandes servidores de BD corporativos como mainframes).

Figura 4.4 Arquitectura Cliente Servidor

Las formas de añadir inteligencia al cliente ha sido a través de Javascript y applets; Javascript es un lenguaje relativamente sencillo, interpretado, cuyo código fuente se introduce en la página HTML por medio de los tags <SCRIPT> …</SCRIPT>; su nombre deriva de una cierta similitud sintáctica con Java y los applets tienen mucha más capacidad de añadir inteligencia a las páginas HTML que se visualizan en el browser, ya que son verdaderas clases de Java (ficheros *.class) que se cargan y se ejecutan en el cliente.

4.4.1 Servlets

Programa que se ejecuta en un servicio de red, (HTTP) que recibe y responde a las peticiones de uno o más clientes. Proporciona las mismas ventajas Java en cuanto a portabilidad y seguridad. Su rendimiento es bastante bueno, una vez que son llamados por primera vez, quedan activos en la memoria del servidor hasta que el programa que controla el servidor los desactiva, con lo que minimiza el tiempo de respuesta.

Características de los Servlets

1. Independientes del servidor utilizado y del S.O, es decir, que a pesar de ser código Java, el servidor puede ser de cualquier otro lenguaje.

2. Pueden llamar a otros servlets, incluso a otros métodos, de forma que se distribuye eficientemente el trabajo. y permiten redireccionar peticiones de servicios a otros servlets (misma máquina o en una remota).

3. Obtienen información acerca del cliente (permitida por HTTP), como dirección IP, puerto utilizado en la llamada, métodos (GET, POST), etc.

4. Permiten utilizar cookies y sesiones, de forma que pueden guardar información específica acerca de un usuario determinado, personalizando la interacción cliente-servidor.

5. Pueden actuar como enlace entre el cliente y una o varias BD en arquitecturas cliente-servidor de 3 capas (si la BD está en otro servidor).

6. Generación dinámica de código HTML dentro de una propia página HTML.

- **Ciclo de vida de un servlet**

Se ejecutan en el servidor HTTP como parte integrante del propio proceso del servidor, por lo que, HTTP es el responsable de la inicialización, llamada y destrucción de cada objeto de un servlet, tal y como se observa en la figura 4.5:

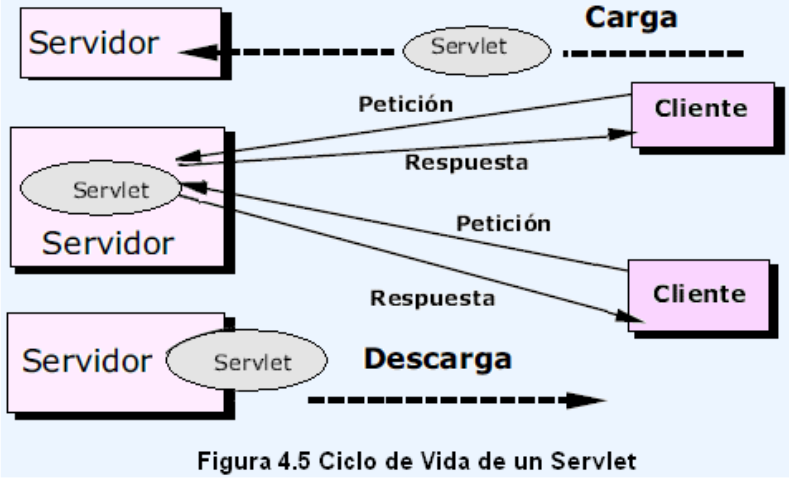


Figura 4.5 Ciclo de Vida de un Servlet

El servidor web se comunica al servlet con métodos de la interface constituida por:

- **Método init**

Se usas al cargar por primera vez un servlet, el método init() es llamado por el servidor HTTP, init no será llamado nunca más mientras el servlet se ejecute, lo que le permite efectuar operaciones costosas de inicialización, ya que sólo se ejecutará la primera vez. init() tiene un argumento único, que referencia a un objeto de la interface ServletConfig, proporcionando argumentos de inicialización del servlet, este objeto dispone del método getServletContext().

- **Método destroy**

Permite que el servlet concluya tareas ordenadamente, con lo que, es posible liberar recursos (archivos abiertos, conexiones con BD, etc.) de forma limpia y segura, cuando no son necesarios para lo que no hace falta redefinir a destroy. A veces sucede que al llamar a destroy existen peticiones de servicios ejecutándose por service, lo que provoca fallas del sistema, es conveniente llamar a destroy de forma que retrase la liberación de recursos hasta que no concluyan las llamadas a service.

Método service

Es el método núcleo del servlet, cada petición por parte del cliente se traduce en una llamada al método service() del servlet, que lee la petición y debe producir una respuesta en base a los dos argumentos que recibe:

- Objeto de la interface **ServletRequest** con datos enviados por el cliente, que incluyen parejas de parámetros clave/valor y un InputStream.

- Objeto de la interface **ServletResponse**, que encapsula la respuesta del servlet al cliente, en el proceso de preparación de la respuesta, es necesario llamar al método setContentType, a fin de establecer el tipo de la respuesta.

Existen 2 formas de recibir información de un formulario HTML en un servlet; en la primera se obtienen los valores de los parámetros (métodos getParameterNames y getParameterValues) y en la segunda se recibe con InputStream o Reader.

Método GET

Se encarga de solicitar información a un servidor Web, al información puede ser un archivo, un programa ejecutado, etc. en la mayoría de los servidores web los servlets son accedidos mediante un URL que comienza por /servlet/. En siguiente ejemplo, GET solicita servicio del MiServlet a miServidor.com (en negritas):

```
GET /servlet/MiServlet?nombre=Sandra&Apellido=Alvarez%20de%20Diaz HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.5 (
compatible;
MSIE 4.01;
Windows NT)
Host: miServidor.com
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg
```

Método POST

POST permite al cliente enviar información al servidor, se debe utilizar en lugar de GET en aquellos casos donde se requiera transferir una gran cantidad de datos (formularios). POST no tiene limitantes como GET en cuanto a volumen de información transferida, pues no va incluida en la URL de petición, sino que viaja encapsulada en un inputstream que llega al servlet a través del teclado (negritas).

```
POST /servlet/MiServlet HTTP/1.1
User-Agent: Mozilla/4.5 (
compatible;
MSIE 4.01;
Windows NT)
Host: www.MiServidor.com
Accept: image/gif, image/x-bitmap, image/jpeg, image/jpeg, */
Content-type: application/x-www-form-urlencoded
Content-length: 39
nombre=Sandra&Apellido=Alvarez%20de%20Diaz
```

4.4.2 Tag Libs

Las bibliotecas Tag o bibliotecas JSP Tag son bibliotecas desarrolladas para ser integradas y utilizadas en los JSP de aplicaciones J2EE, las bibliotecas ejecutan acciones utilizadas en el JSP en forma de etiquetas XML, dichas acciones manipulan datos y variables del JSP y de la aplicación JAVA J2EE. La TagLib es definida por un descriptor de taglib o Tag Librarie Descriptor y clases Java que implementan la interfaz JSP Tag.

El descriptor se representa por un archivo XML de extensión tld que describe las relaciones entre etiquetas y clases Java, la etiqueta XML escrita en el JSP recurre a estas acciones y son reemplazadas únicamente durante la compilación del JSP en el servidor de aplicación por llamado de las clases correspondientes. Por ejemplo:

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>
```

Para optimizar el código de las aplicaciones J2EE, varios frameworks utilizan taglibs, como Struts, Spring MVC, JSTL, etc. En JSTL existe una biblioteca estándar para la mayoría de funcionalidades básicas de plicaciónes J2EE, y la mayoría de taglibs presentes extienden de la biblioteca JSTL.

- **Creación de acciones personalizadas en JSP**

En la librería core de JSLT se tiene la mayoría de las herramientas necesarias para las aplicaciones, pero existen casos que no serán suficientes o que no interese crear etiquetas propias en las aplicaciones. A continuación algunos pasos para crearlas:

Implementación de la clase manejadora

Por cada tag, se debe de definir su clase manejadora que implemente las operaciones a realizar por la aplicación. La interfaz javax.servlet.jsp.tagext.Tag es la que proporciona el soporte para crear la clase manejadora con los elementos mínimos necesarios. Los métodos ejecutados durante la acción son:

void setPageContext(PageContext pc): Primer método invocado por el contenedor y permite establecer el contexto de la página actual.
void setParent (Tag parent): Segundo método ejecutado en la acción y al que se le pasa el objeto Tag o en su defecto Null.
int doStartTag(): Se ejecuta una vez que el contenedor detecta la etiqueta de comienzo de la acción.
int doEndTag(): Se invoca cuando detecta la etiqueta de cierre de la acción.

Crear el archivo de librería

Este tendrá la información necesaria para que el contenedor web pueda interpretar la información, es un XML con extensión .tld, donde se indica el conjunto de acciones que pertenecen a la librería, como:

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN" "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

Luego, se indica el elemento raíz taglib y dentro las etiquetas del documento como:

jsp-version: Versión de la especificación JSP utilizada por la librería.
short-name: posible valor elegido para ser utilizado como prefijo.
uri: es la URL asociada a la librería.
description: Texto descriptivo de la librería.
tag: definición de librería que puede contener los siguientes subelementos:
- **name:** Nombre de la acción.
- **body-content:** determina cómo se evalúa el cuerpo de la acción.
- **description:** Descripción de la acción.
- **name.** Nombre del atributo
- **required.** Indica si el atributo es o no obligatorio
- **type.** Tipo de dato del atributo (nombre de clase).

Por ejemplo:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/webjsptaglibrary_1_2.dtd">
<taglib>
<tlib-version>1.0</tlib-version>
<jsp-version>1.2</jsp-version>
 <short-name>ejemplo</short-name>          <uri>http://www.milibreria.tld</uri>
<description>Ejemplo de librería</description>
<tag>
<name>saludo</name>
<tag-class>ejemplo.Saludo</tag-class> <body-content>empty</body-content>
<description>tag de ejemplo</description> </tag>
 </taglib>
```

Utilización

Para distribuir las clases manejadoras se trata el archivo .class o se genera un .jar en el que se incluyen todas las clases. En ambos casos, las clases deben ser accesibles desde la aplicación Web, y proceder según la distribución de las clases:

Clases.class: los archivos se incluyen en un paquete en WEB-INF\classes.
Clases distribuidas .jar: debe incluirse en WEBINF\lib de la aplicación.

En la librería .tld, debe incluirse en el .jar junto con las clases manejadoras dentro de WEB-INF. Para poder usar las acciones en cualquier JSP de la aplicación se tiene que incluir una referencia a la librería a través de la directiva taglib.

Por ejemplo para utilizar una librería ya creada, al principio del JSP se debe poner la siguiente línea, sustituyendo el uri, por el que se haya creado:

<%@ taglib uri="http://www.ejemplolibreria.tld" prefix= "ejemplo"%>

4.5 J2EE (Java 2 Enterprise Edition).

Es la plataforma de tecnología Java más avanzada, que provee un rango completo de funcionalidades empresariales para el desarrollo de aplicaciones tanto de computadoras como de servidores. Ha sido diseñada para proporcionar un ambiente integrado en la creación de programas Java a todo nivel requerido por los usuarios.

J2EE se considera un estándar de desarrollo de aplicaciones empresariales multicapa, pues simplifica las aplicaciones basándolas en componentes modulares y estandarizados, administra servicios para los componentes, y maneja funciones de forma automática, sin necesidad de programación compleja. J2EE contiene (figua 4.6):

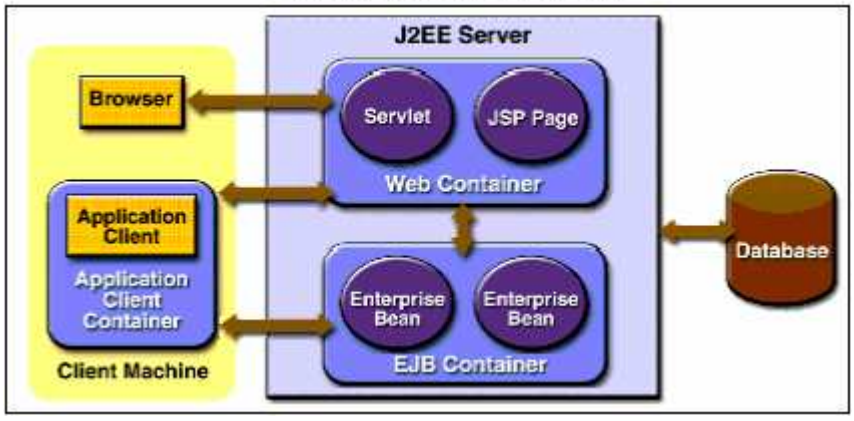
- **Presentación lado del Cliente (Cient - Side Presentation):** En esta área se encuentra el rango de funcionalidades que J2EE ha de ejecutar en la máquina, donde el cliente accede a las herramientas. En este grupo se encuentran páginas Web (HTML), Applets y aplicaciones de escritorio.

- **Presentación lado Servidor (Server - Side Presentation):** Incluye aplicaciones Java desligadas del cliente ejecutándolas en el servidor y agiliza el despliegue. Aplicación Web, JSP, Servlets, código XML y servicios Web J2EE.

- **Lógica de Negocios lado Servidor (Server – Side Business Logia):** No necesariamente son aplicaciones, sino dispositivos encargados de generar procesos de modo no visible para el usuario (EJB y JavaBeans), que permiten la interacción de componentes Web Services entre las distintas capas. EJB container ejecuta Enterprise Beans en aplicaciones del lado del servidor.

- **Información de Sistema (Enterprise Information System):** Aquí se incluyen datos guardados necesarios para la ejecución de los componentes J2EE.

Figura 4.6 Arquitectura J2EE



La funcionalidad de J2EE es separada entre sus capas, de tal modo que, permite segregación de responsabilidades, reusabilidad y escalabilidad entre sus beneficios. La separación puede ser física y cada capa se ubicaría en diferentes unidades de hardware o simplemente la separación puede ser lógica.

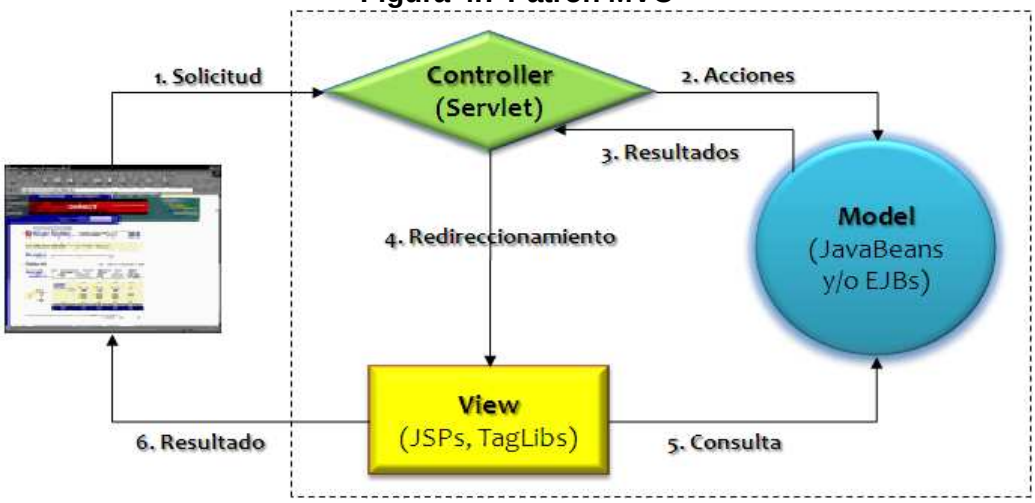
4.5.1 Struts

Framework con un conjunto de clases e interfaces que cooperan para solucionar un problema específico de software, sus principales características son:

- Tiene múltiples clases, cada una provee abstracción de determinado concepto.
- Define cómo esas abstracciones trabajan juntas para solucionar el problema.
- Sus componentes son reutilizables y organiza patrones a alto nivel.
- Provee comportamiento genérico para que varios tipos de aplicaciones lo usen.

Struts, implementa el patrón de arquitectura MVC (Model-View-Controller) que es el que define la organización independiente del Modelo (Objeto de Negocio), la Vista (interfaz con usuarios) y el Controlador (controlador del flujo de la aplicación).

Figura 4.7 Patrón MVC



Modelo: Aquí se conocen los datos a mostrar y es donde se consideran todas las operaciones que se aplican para transformar el objeto. Representan los datos de la aplicación, los datos empresariales y las reglas de negocio que gobiernan el acceso y la actualización de tales datos. El modelo no es consciente de los datos de presentación ni de cómo esos datos serán mostrados en el navegador.

Vista: Es la presentación de la aplicación, usa los datos de consulta del modelo para obtener el contenido y es independiente de la lógica de la aplicación, ya que se mantiene igual si hay modificaciones en la lógica de negocio, es decir, la vista debe mantener su consistencia de presentación aun cuando el modelo cambie.

Controlador: Cuando el usuario manda una petición, esta viaja a través del controlador, el que es responsable de interceptar las peticiones desde la vista y pasarlas al modelo para que se lleve a cabo la acción adecuada. Una realizada la acción sobre los datos, el controlador es responsable de dirigir a la vista apropiada.

Funcionamiento MVC. (Figura 4.8)

- **Modelo.** Proporciona la lógica de negocio que suministra interfaces a la BD. (Generalmente clases java), no existe un formato específico para realizarlo, lo que permite reutilizar código en otros proyectos. Sus componentes son:
  Action Beans que se encargan de Obtener valores de Action Form, JavaBean, request, sesión, etc., llamar a objetos de negocio del Modelo y analizar resultados para retornar el ActionForward adecuado.

  System State Beans: Objetos de negocio, que representan el estado actual del sistema, (un carrito de compras que el usuario modifica en su interacción con la aplicación) pueden ser: JavaBeans o EJBs de los que se guarda referencia en la sesión del usuario, se modifican en Action y se consultan en JSPs.

  BusinessLogic Beans: Objetos que implementan la lógica de negocio, el cómo hacer las cosas y su propia persistencia.

- **Vista.** Responsable de presentar la información a los usuarios y aceptar sus entradas de información suministrada por el modelo, principalmente se emplean JSP para presentarlos y amplían sus capacidades con etiquetas javaScript.

  Internacionalización: Maneja un archivo con información del idioma (.properties), donde se encuentran clases que contienen claves y valores con el formato de texto e idioma principal. Struts por defecto asigna a cada usuario el idioma principal de la aplicación, que se puede cambiar con una lista de idiomas:

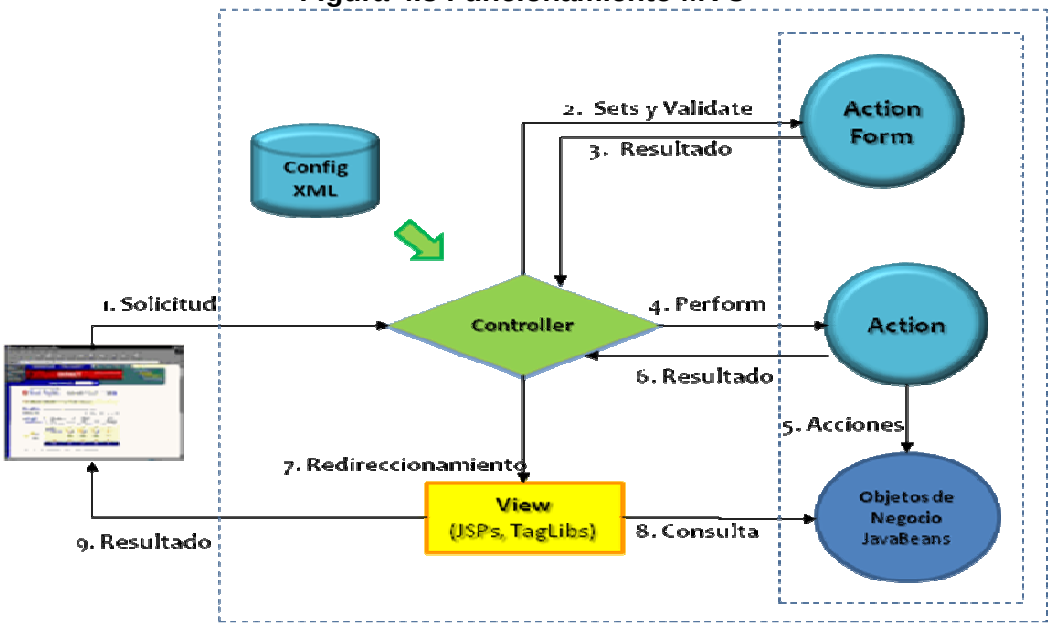
  ```
  session.setAttribute(Action.LOCALE_KEY, new java.util.Locale(country, language))
  ```

- **Controlador.** Funcionalidad, desde que un usuario genera un estímulo (click en un link, envío de un formulario, etc.) hasta generar la interfaz de respuesta. El proceso, llama objetos de negocio para resolver la funcionalidad de la lógica y según el resultado, ejecutar un JSP que genera la interfaz resultante. Struts incluye un servlet a partir de la configurar struts-config.xml, recibe solicitudes del usuario, llama al ActionBean y, según su retorno, ejecuta un JSP.

Para escribir un ActionForm hay que tener en cuente los siguientes principios:

- No debe tener nada de la lógica de negocio.
- Tener implementación de getters y setters (un par por cada input del form; por ejemplo getNombre y setNombre) y los métodos reset y valídate.
- Filtro entre el usuario y el Action, deteniendo errores de inconsistencia.
- Si el formulario se desarrolla en varias páginas el ActionForm y el Action deberán ser los mismos, lo que permitirá, que los input se reorganicen.

Figura 4.8 Funcionamiento MVC



4.5.2 Java Server Faces.

Framework para aplicaciones Web, simplifica el diseño de la interfaz de usuario de la aplicación, separa la presentación Web, de la lógica comercial, sus elementos son:

- Conjunto de API´s para representar componentes gráficos y manejar estados.
- Controlar eventos y da soporte a la internacionalización.
- Validar la entrada de datos.
- Definir la navegación por las distintas páginas de la aplicación.
- Interfaces accesibles para todo el mundo.
- Librería JSP personalizada para expresar una interfaz JSF en una página JSP.

Beneficios de JavaServer Faces

- Facilidad de empleo y separación de la lógica y de la presentación.
- Facilidad de conectar la capa de presentación con el código de la aplicación.
- Facilidad de integración, aunque se estén desarrollando por separado.
- Extiende componentes de clases para generar otros para clientes específicos.

Tecnología que posee JSF

- Protocolo de Transferencia de Hipertexto (http)
- Servlets, JavaBeans, JSP.

Componente de interfaz de usuario (UI): Objeto mantenido por el servidor, que da una funcionalidad específica para interactuar con el usuario. Estos componentes son JavaBeans con propiedades, métodos y eventos, organizados dentro de una vista de árbol de componentes mostrados como una página.

Renderer: Muestra los componentes de la interfaz de usuario, y traduce la entrada del usuario a valores de los componentes, éstos pueden ser diseñados para trabajar con uno o más componentes de interfaz de usuario que a su vez puede asociarse con renders distintos.

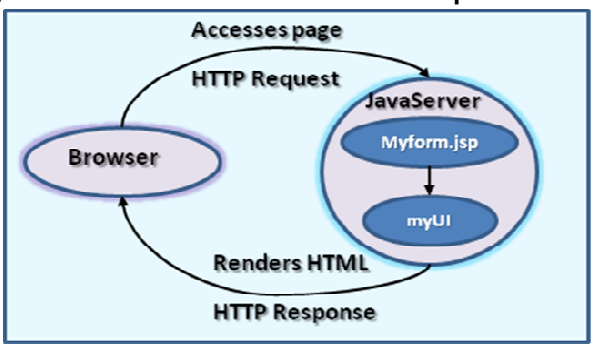
JSF es orientado a eventos, lo que permite a las aplicaciones atarse menos a HTTP y simplificar el esfuerzo del desarrollo. Mueve fácilmente la capa de negocio y presentación fuera del controlador y saca la lógica de negocio del JSP, ya que, las clases controladoras no se atan a JSF, lo que permite que se ejecuten test sobre ellas.

El Bean controlador de mapeo media entre la vista y el modelo, por ello es importante imitar la lógica de negocio y la lógica de persistencia en el bean ligado a JSF. Es común delegar la lógica de negocio al modelo de la aplicación; en este caso, el bean de control mapea objetos del modelo y la vista pude ser mostrada como propiedad de bean.

Por ejemplo: La interface de usuario se crea con JSF (myUI) se ejecuta en el servidor y se renderiza en el cliente. El JSP, myform.jsp, dibuja los componentes de la interface con etiquetas personalizadas de JSF. El UI maneja los objetos referenciados por el JSP **(figura 4.9)**:

- Los objetos componentes que mapean las etiquetas sobre el JSP.
- Oyentes de eventos, validadores y conversores en los componentes.
- Objetos del modelo que encapsulan datos y funcionalidades de componentes.

Figura 4.9 Funcionamiento de los Componentes JSF



JSF introduce el concepto de bean administrado, que es el pegamento de la lógica de la aplicación, los beans se definen en el archivo faces-config.xml y dan acceso total a los métodos mapeados. Su característica es poder crear beans de respaldo y Data Access Objects (DAO), el bean de respaldo es un POJO sin dependencia de la implementación.

El controlador de JSF es FacesServlet y no es consciente de que acción tomar, solo es consciente del resultado y utiliza eso para decidir sobre la navegación. El servlet es el consciente de la acción o método a llamar en cada evento de usuario. UIComponents es fundamento de la vista y representa el comportamiento de la aplicación.

4.5.3 Hibernate.

Hibernate es una capa de persistencia objeto/relacional y un generador de sentencias SQL, permite diseñar objetos persistentes que incluyen polimorfismo, relaciones, colecciones y un gran número de tipos de datos; de manera muy rápida y optimizada se generan BD de cualquier tipo: Oracle, DB2, MySql, etc..

Si se trabaja con POO y Bases de Datos relacionales, seguramente existirán diferencias, ya que el modelo relacional trata con tuplas, relaciones y conjuntos. Sin embargo, la POO trata con objetos, atributos y relaciones. Pero al requerir que los objetos sean persistentes utilizando una BD, se observan discrepancias entre ambos, denominada diferencia objeto-relacional. Un mapeador objeto-relacional (ORM) ayudará a evitar esta diferencia.

Si se requiere que lo objetos sean persistentes, se abre una conexión ODBC, se crea una sentencia SQL y se copian todos los valores de las propiedades sobre un PreparedStatement. Esto es fácil para un value object:VO pequeño pero no funciona para un objeto con mas propiedades. Este no es el único problema, ya que enfrentamos otras incógnitas como saber ¿Qué pasa con las asociaciones? ¿Qué hacer si el objeto contiene otros objetos?, el almacenamiento de la Base de Datos, etc., preguntas que surgen al cargar los datos en la BD y en el VO (value object (VO) información de negocio). Se necesita un ORM (Object Relational Mapping).

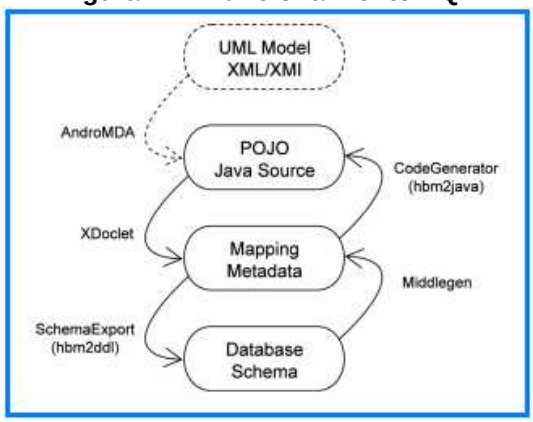
ORM hace todas las tareas pesadas, sólo se debe definir la forma de establecer la correspondencia entre las clases y las tablas (indicando que propiedad corresponde con que columna y que clase con que tabla, etc.). Después se utilizan POJO's (Plain Old Java Objects) y se le dice a la ORM que los haga persistentes, con una instrucción: orm.save(myObject) y podrá leer o escribir en BD usando VO's (**figura 4.10)**.

El modelo de dominio representa entidades de negocio utilizadas en la aplicación. En la arquitectura en capas, el modelo se utiliza para ejecutar la lógica de negocio, se comunica con la de persistencia para recuperar y almacenar objetos persistentes.

Figura 4.10 Elementos Hibernate

Hibernate Query Language HQL: Hibernate tiene un lenguaje para manejar consultas a BD, es similar a SQL y es utilizado para obtener objetos de la BD según las condiciones especificadas en HQL, permite emplear un lenguaje intermedio según la BD; el lenguaje especificado será traducido a SQL dependiente de cada base de datos de forma automática y transparente.

Figura 4.11 Funcionamiento HQL



Conceptos básicos de Hibernate

Para almacenar y recuperar objetos de la BD, se debe mantener una conversación con el motor de Hibernate mediante un objeto especial, (el más importante) que es la Sesión (Session). Esto se equipara al concepto de conexión JDBC y cumple un papel muy parecido, que, delimita operaciones relacionadas con el proceso de negocio, marca una transacción y aporta otros servicios como caché de objetos que evita interacciones innecesarias con la BD.

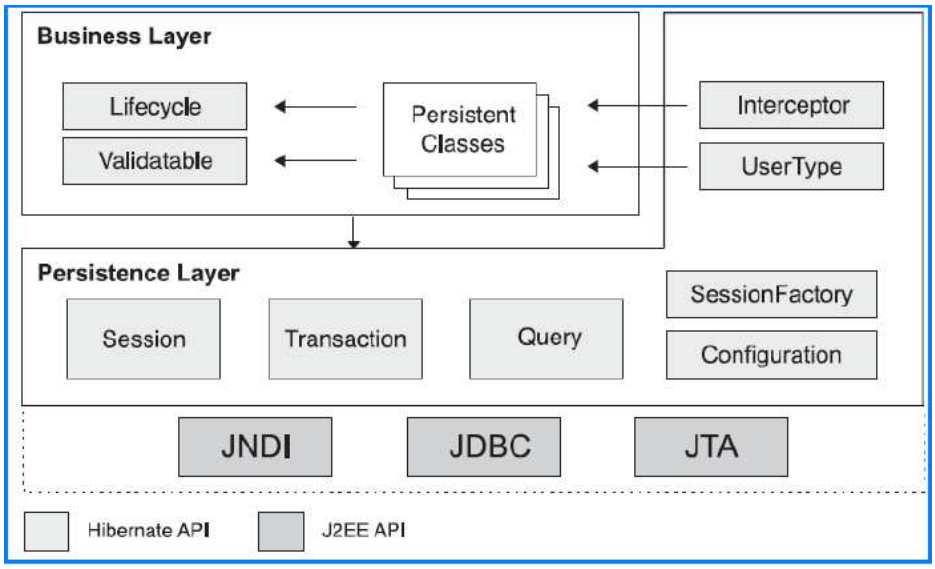
Session ofrece métodos como save(Object object), createQuery(String queryString), beginTransaction(), close(), etc. para interactuar con la BD tal como se acostumbra con una conexión ODBC, pero con una diferencia: mayor simplicidad para guardar un objeto, por ejemplo, session.save(miObjecto), sin necesidad de especificar una sentencia SQL.

Lo que introduce a los conceptos de transient y persistent: los primeros son objetos que sólo existen en memoria y no en un almacén de datos, en algunos casos, no se almacenan en la BD. Los segundos ya han sido almacenados, por tanto son objetos persistentes. Los objetos transient han sido instanciados sin haberse almacenado en sesión, los objetos persistentes han sido creados y almacenados en sesión o devueltos en una consulta a través de la sesión.

Igual que con conexiones ODBC se crea y cierra sesión, aunque no hay una relación 1:1 entre sesiones y conexiones, es decir, no se tiene que abrir y cerrar simultáneamente sesiones y conexiones ODBC, la política a seguir dependerá del contexto del proceso de negocio de cada situación, donde Hibernate posee amplias posibilidades siendo solamente necesario crear y cerrar explícitamente las sesiones de Hibernate.

Arquitectura

El API de Hibernate es una arquitectura de dos capas (Capa de persistencia y Capa de Negocio). A continuación se muestran los roles de las interfaces Hibernate más importantes en las capas de persistencia y de negocio de una aplicación J2EE. La capa de negocio se situada sobre la capa de persistencia, ya que actúa como un cliente de la persistencia.



Las Interfaces se clasifican de la siguiente forma:

- Interfaces llamadas por la aplicación para realizar operaciones básicas:
  - **Session**: interfaz primaria en cualquier aplicación Hibernate (SessionFactory).
  - **Transaction**
  - **Query**: permite realizar peticiones a la BD y controla cómo se ejecuta dicha petición (query). Las peticiones se escriben en HQL o en el dialecto SQL nativo de la BD utilizada. El Query se utiliza para enlazar parámetros de la petición, limitar el número de resultados devueltos por la petición y para ejecutarla.
- Interfaces llamadas por el código de la infraestructura para configurar Hibernate. La más importante es Configuration: se utiliza para configurar y arrancar Hibernate. y especifica la ubicación de los documentos que indican el mapeado de objetos y propiedades Hibernate, y a continuación crea la Session Factory.

- Interfaces callback, permiten a la aplicación reaccionar en determinados eventos dentro de la aplicación, tales como Interceptor, Lifecycle, y Validatable.

- Interfaces que permiten extender funcionalidades de mapeado de Hibernate, como por ejemplo UserType, CompositeUserType, e IdentifierGenerator.

5. PRUEBAS

La etapa de pruebas de software es un elemento crítico, ya que mediante esta etapa se determina la garantía de calidad del software desarrollado, las pruebas implican los siguientes puntos:

- Verificar que todos los requisitos se han implementado correctamente.
- Verificar la integración adecuada de los componentes.
- Identificar y asegurar que los defectos encontrados se han corregido antes de entregar el software al cliente final.
- Diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

Las pruebas no son una actividad sencilla, no es una etapa del proyecto en la cual se asegura la calidad, sino que las pruebas deben ocurrir durante todo el ciclo de vida del proyecto para minimizar errores, se puede probar la funcionalidad desde los primeros prototipos; probar la estabilidad, cobertura y rendimiento de la arquitectura; hasta llegar a probar el producto final. Lo que conduce al principal beneficio de la etapa de pruebas, que es proporcionar reacción mientras todavía hay tiempo y recursos para hacer algo.

Es un proceso que se enfoca sobre la lógica interna del software y las funciones externas, es decir es un proceso de ejecución de un programa con la intención de descubrir un error. Un buen caso de prueba es aquel que tiene alta probabilidad de mostrar un error no descubierto hasta entonces, la prueba tiene éxito si descubren errores que hasta ese momento han pasado desapercibidos. En realidad las pruebas no pueden asegurar la ausencia de defectos; si no que sólo demuestran que existen defectos en el software.

5.1 Pruebas unitarias:

Su objetivo es la de verificar aisladamente cada método implementado en un módulo o clase. Sin embargo, no se puede olvidar que dichas clases conforman un sistema estrechamente interrelacionado, de manera que entre una y otra puede haber relaciones de colaboración y/o dependencia. La idea es escribir casos de prueba para cada función no trivial o método en el módulo de forma que cada caso sea independiente del resto. Algunas de las características a cumplir son:

- **Automatizable**: no se debe requerir de la intervención manual.
- **Completas**: deben cubrir la mayor cantidad de código.
- **Reutilizables**: crear pruebas que puedan ejecutarse más de una vez.
- **Independientes**: la ejecución de una prueba no debe afectar a otra.
- **Profesionales**: deben considerarse igual que el código, con la misma profesionalidad, documentación, etc.

Aunque las características descritas no tienen que ser cumplidos al pie de la letra, es recomendable seguirlas ya que de no hacerlo las prueban pierden parte de su función, que es aislar cada parte del programa y mostrar que las partes individuales son correctas, a continuación algunas de las ventajas de estas pruebas aisladas:

- **Fomentan el cambio**: Ya que facilitan que el programador cambie el código para mejorar su estructura (refactorización), ya que permite hacer pruebas sobre los cambios, asegurando que éstos no han introducido errores.

- **Simplifica la integración**: Permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.

- **Documenta el código**: Las propias pruebas son documentación del código debido a que ahí se puede ver cómo utilizarlo.

- **Separación de la interfaz y la implementación**: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces, se puede cambiar cualquiera de los dos sin afectar al otro, usando a veces objetos para simular el comportamiento de otros más complejos.

- **Se acotan errores y son más fáciles de localizar**: debido a que las pruebas unitarias pueden descubrirlos.

En Java existe una herramienta open source para el desarrollo de pruebas unitarias, que es JUnit, este framework se extiende con otros más para llevar a cabo la realización de pruebas unitarias más específicas, insertándose como plugins en los entornos de desarrollo.

Todas las aplicaciones Web se ejecutan dentro de un servidor Web o en un servidor de aplicaciones, estos elementos son el contenedor de la aplicación, y como su nombre lo indica contiene los recursos, datos y objetos que la forman. Hay diferentes maneras de realizar pruebas unitarias sobre una aplicación Web:

- Pruebas desde el interior del contenedor de la aplicación; en este caso se instala un módulo dentro del contenedor de la aplicación (servidor Web o de aplicaciones), que le dota de esta capacidad.

- Pruebas desde el exterior del contenedor de la aplicación; para este caso, mediante aplicaciones externas a la aplicación web, se ejecutan pruebas contra la aplicación, que normalmente son peticiones de páginas o servicios, simulando la interacción de los usuarios.

Para realizar estas pruebas se cuenta con herramientas open source como:

- **Httpunit.** Extensión de junit para realizar pruebas unitarias sobre HTTP.
- **Htmlunit.** Extensión de junit para realizar pruebas unitarias web. Se basa en obtener los elementos que componen la página web devuelta por el servidor.

- **Seleniuum.** Extensión de htmlunit, su principal diferencia es que impregna un cliente web real. Se instala como un plugin sobre Firefox o Internet Explorer, y permite trabajar con ellos como cliente web de pruebas.

- **Jwebunit.** Framework de pruebas unitarias web basado en htmlunit.

- **Shale.** Es un framework para realizar pruebas unitarias en aplicaciones JSF.

- **Cactus.** Framework para realizar pruebas unitarias de aplicaciones web, puede realizar pruebas sobre servlets y páginas jsp. Diseñado para probar aplicaciones con el patrón MVC, soporta servlets, clases, taglibs, filters etc.

Para realizar un conjunto de pruebas unitarias, las partes fundamentales son:

- La interfaz de la aplicación, incluye navegación y comprobando que la aplicación cumple los Casos de Uso, al realizar la navegación correcta por las diferentes páginas y mostrando la información correcta al usuario, se prueba que se comporta correctamente al realizar acciones incorrectas.

- La lógica de la aplicación; una vez que la aplicación se comporta externamente de manera correcta, hay que comprobar que internamente también se comporta correctamente. Asimismo, se debe validar, los diferentes estados y la persistencia en base de datos, como elementos frecuentes.

Es importante darse cuenta que las pruebas unitarias no descubrirán todos los errores del código, ya que, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, es importante anticipar todos los casos especiales de entradas que puede recibir la unidad de programa en estudio.

5.2 Pruebas de integración

Se realizan en el desarrollo de software, una vez que se han hecho todas las pruebas unitarias, se deben probar todos los elementos unitarios que componen el proceso. Y se espera es que al realizar las pruebas de un gran conjunto de partes, estas funcionen juntas. Su necesidad de realización es porque los módulos que forman el programa suelen fallar cuando trabajan de forma conjunta, aunque previamente se demuestre que funcionan individualmente. Por ello realizar dichas de pruebas, asegura que los módulos relacionados se ejecutan correctamente.

Con esta práctica se consigue formar el sistema global a medida que se comprueba como los componentes interaccionan y se comunican libres de errores. Para realizar la integración del sistema se puede tomar la decisión de poner en práctica la integración ascendente, es decir, comenzar por los módulos más bajos hasta llegar a la Clase principal y sucesivamente hasta alcanzar el sistema en general.

Las pruebas de integración consisten en detectar las fallas de interacción entre las distintas clases que componen al sistema. Los datos se pueden perder en una interfaz, ya que, un módulo puede tener un efecto adverso e inadvertido sobre otro, al combinar las subtareas, pueden no producir la tarea principal deseada; la imprecisión en las pruebas individuales puede crecer hasta niveles inaceptables; y las estructuras de datos globales pueden presentar problemas.

Lo anterior se deduce debido a que cada clase probada unitariamente se inserta de manera progresiva dentro de la estructura, además de que a la par se van siguiendo los lineamientos dictados por el diseño, las pruebas de integración son realmente el mecanismo para comprobar el correcto ensamblaje del sistema completo. Al efectuar la integración de los módulos, se debe concentrar la búsqueda de defectos como aquellos que puedan provocar las excepciones arrojadas por los métodos; el empleo de operaciones e invocación inadecuada de los métodos.

Para realizar pruebas de integración en sistemas Orientados a Objetos se cuenta con estrategias diferentes. Una de ellas es la **integración por hilos** (integración por reacción directa o indirecta a un evento); consiste en reunir el conjunto de clases requerido para responder a la cadena de mensajes provocada por un evento del sistema, para luego integrarlas y probarlas individualmente.

Otra estrategia es la **integración por dependencia de la clase**, en la cual se comienza el ensamble del sistema probando las clases independientes, que usan muy poca o ninguna otra clase del sistema; después se continúa con la prueba de las clases dependientes, esta secuencia de pruebas de clases dependientes continúa hasta que el sistema entero se completa.

- **Pruebas de rendimiento:**

Los usuarios esperan un alto grado de servicio y también esperan que mantenga constante sin tener en cuenta las circunstancias objetivo de las pruebas de rendimiento y se ejecutan para determinar la respuesta un sistema ante una cierta carga y para validar atributos relacionados con la calidad, escalabilidad o el uso de recursos. Es básico para el rendimiento del sistema, que las pruebas comiencen con el desarrollo y se extiendan hasta el paso a producción, ya que cuanto más tarde se descubra un problema de rendimiento, más alto será el costo en remediarlo.

- **Pruebas de Carga**

Tales pruebas se ejecutan para comprender el comportamiento de una aplicación ante una carga determinada. Esta carga puede ser el número de usuarios esperado ejecutando o un número de transacciones durante un tiempo determinado, el resultado de estas nos dará el tiempo de respuesta de todas las transacciones críticas. Si la base de datos, servidor de aplicación también se monitorizan, entonces este test puede mostrar potenciales problemas de tráfico en la aplicación.

- **Pruebas de Estrés**

Son utilizadas normalmente para someter a la aplicación al límite de su funcionamiento, mediante la ejecución de un número de usuarios muy superior al esperado, con la finalidad de determinar la robustez de la aplicación cuando la carga es extrema; además de que ayuda a los administradores a determinar si la aplicación se comportara correctamente en dichas situaciones. Adicional a lo anterior, también se puede determinar el límite real de la aplicación en cuanto a número de usuarios concurrentes, numero de transacciones por segundo, etc...

- **Pruebas de Resistencia (SOAK)**

Dicha prueba consiste en determinar si la aplicación puede mantener la carga esperada de manera continúa y durante un largo tiempo.

- **Pruebas de Picos**

Estas pruebas se realizan insertando carga en el sistema en forma de “picos” que se irán lanzando en distintos momentos de la prueba y que permitirán comprender el comportamiento de la aplicación ante cambios bruscos de carga**.**

5.3 Pruebas de Usuario

Estas pruebas consisten en que el usuario realice tareas concretas en el sistema, midiendo el tiempo que le lleva acabarlas, los errores que comete durante proceso, la facilidad de la navegación, lo entendible que sea el flujo de los datos, entre otras más, que debe cumplir el sistema desarrollado a fin de satisfacer las necesidades del Usuario final, para el que fue construido. La **usabilidad** de un sistema, precisamente mide cómo un usuario verdaderamente navega, busca información e interactúa con el sistema.

Usabilidad: es un anglicismo que significa "facilidad de uso" que proviene de la expresión "user friendly", que poco a poco se ha ido reemplazando debido a sus connotaciones vagas y subjetivas.

El estándar ISO define en grandes rasgos a la usabilidad como, el grado de eficacia, eficiencia y satisfacción con la que usuarios específicos pueden lograr objetivos específicos, en contextos de uso específicos. En consecuencia podemos observar que la usabilidad se compone de dos tipos de atributos:

- **Atributos cuantificables de forma objetiva:** como son la eficacia o número de errores cometidos por el usuario durante la realización de una tarea, y eficiencia o tiempo empleado por el usuario para la consecución de una tarea.

- **Atributos cuantificables de forma subjetiva**: como es la satisfacción de uso, medible a través de la interrogación al usuario, y que tiene una estrecha relación con el concepto de Usabilidad Percibida.

Como se indica, la usabilidad de la aplicación debe entenderse siempre en relación con la forma y condiciones de uso por parte de sus usuarios, así como con las características y necesidades propias de estos. Usualmente toda aplicación se diseña con la intención de satisfacer las necesidades de un grupo concreto y determinado, por lo que esta será más usable cuanto más adaptado esté el diseño al grupo específico, y por tanto menos lo esté para el resto de personas.

Un concepto ligado al de usabilidad es el de accesibilidad, que ya no se refiere a la facilidad de uso, sino a la posibilidad de acceso, es decir, el diseño como prerrequisito para ser usable, posibilite el acceso a todos sus usuarios potenciales, sin excluir aquellos a ninguno.

Pero como saber cuándo realizar las pruebas de Usabilidad, la respuesta es sencilla cuanto antes mejor. La realización de pruebas de usabilidad en una fase inicial del desarrollo no sólo resulta más barata a largo plazo, sino que también se adapta más al plan de trabajo, así como al equipo y al presupuesto. Si se hacen pruebas pronto, es muy fácil identificar los problemas y realizar cambios oportunos antes de construir estructuras sobre los errores.

Es esencial prever tiempos y no esperar a poco antes del lanzamiento, ya que la fecha estará muy próxima y el objetivo sería lanzar a producción y no añadir más procesos, por lo que se recomienda llevar a cabo unas pocas durante las fases de creación de prototipos ya que requieren de poco tiempo y presupuesto para su ejecución y se tendrá más certeza de que lo desarrollado va por buen camino.

1. En las etapas tempranas del proceso de desarrollo, el test de una versión previa o del producto proporciona referencias muy útiles al equipo de diseño.
2. En las etapas medias, el test valida el diseño e informa sobre las posibilidades de refinamiento del mismo.
3. En etapas posteriores, el test ratifica que el producto alcanza los objetivos del diseño.

6. IMPLEMENTACIÓN

Etapa final del ciclo de vida en la aplicación, es la conversión del diseño en una implementación efectiva**, l**a transición es relativamente sencilla ya que las decisiones más complejas en cuanto al diseño ya han sido tomadas. La implementación debe ser correcta, debe satisfacer requerimientos**,** debe ser económica y no debe hacer uso excesivo de los recursos ni exceder el tiempo y el almacenamiento.

El resultado de la fase de implementación es el software que satisface los requisitos funcionales y no funcionales, esta epata se encuentra dividido en tres partes: codificación, desempeño y revisión.

1. **Codificación**: En esta etapa existen cuatro elementos a traducir: El ciclo de vida, descripción de clases, el contenido de métodos y el diccionario de datos.

2. **Desempeño**: El desempeño de una aplicación debe ser considerado desde el análisis, diseño y proceso de implementación. No es necesario ser obsesivo con respecto al mismo, si la aplicación carece de velocidad se debe examinar cada componente de forma individual para saber en dónde focalizar los esfuerzos. La depuración de partes individuales puede beneficiar a la aplicación cuando ésta se ejecute de forme integrada.

   Los sistemas pueden ser eficientes más no perfectos, se puede llegar a una condición ideal en el cual la aplicación sea lo suficientemente rápida que satisfaga la demanda de muchos usuarios en cuánto a velocidad y lo suficientemente robusta para que soporte inserciones masivas y/o procesos complejos. Todo esto es un proceso de suficientes recursos a nivel de hardware (memoria, espacio y conectividad) y uso adecuado de los mismos.

3. **Revisión**: Una vez que el código ha sido producido, este debe ser revisado. Las inspecciones requieren que el código sea leído y entendido por personas distintas a sus autores. Se revisa el comportamiento actual del sistema o partes del sistema, contra los requerimientos y especificaciones. El objetivo de esto es detectar errores antes de que éstos entren en producción.

6.1 Servidores

Un servidor es una maquina remota que almacenan información en forma de páginas web y a través del protocolo HTTP lo entregan a petición de los clientes (navegadores web) en formato HTML, el término también puede utilizarse para referirse a la computadora física en la cual funciona el software, una máquina cuyo propósito es proveer datos de modo que otras máquinas puedan utilizarlos.

Estos significados pueden llevar a confusión, ya que, en el caso de un servidor web, este término podría referirse a la máquina que almacena y maneja los sitios web, y en ese sentido se utiliza por compañías que ofrecen hosting u hospedaje. Por otro lado podría referirse, al servidor, como el servidor http, que funciona en la máquina y maneja la entrega de los componentes de páginas web en respuesta a peticiones de los navegadores de los clientes.

En este sentido, los archivos para cada sitio de Internet se almacenan y se ejecutan en el servidor, en la actualidad existen muchos servidores en Internet y varios tipos de servidores y su función común es la de proporcionar el acceso a los archivos y servicios. Un servidor sirve información a las maquinas que se conecten a él, pudiendo acceder a programas, archivos y otra información dentro del servidor.

Para el caso de Servidores Web, es una máquina que usa el protocolo http para enviar páginas Web a la máquina de un usuario cuando este las solicita, existen servidores web, servidores de correo y servidores de bases de datos siendo solamente algunos a los que tiene acceso la mayoría de la gente al usar Internet. Algunos servidores manejan solamente correo o solamente archivos, mientras que otros hacen más de un trabajo, ya que una misma máquina puede tener diferentes programas de servidor funcionando al mismo tiempo. Todos los servidores se conectan a la red mediante una interfaz que puede ser una red o mediante conexión remota. A continuación los diversos tipos de servidores del mercado actual:

- **Plataformas de Servidor (***Server Platforms***):** usado como sinónimo de sistema operativo, la plataforma es el hardware o software subyacentes para un sistema, es decir, el motor que dirige el servidor.

- **Servidor de Aplicaciones** (Application Servers**):** usado como middleware (software que conecta dos aplicaciones), estos ocupan gran parte del territorio entre servidores de bases de datos, el usuario, y la conexión.

- **Servidores de Audio/Video (***Audio/Video Servers***):** añaden capacidades multimedia a los sitios web permitiéndoles mostrar contenido multimedia en forma de flujo continuo desde el servidor.

- **Servidores de Chat (***Chat Servers***):** permiten intercambiar información a gran cantidad de usuarios con la posibilidad de realizar discusiones en tiempo real.

- **Servidores de Fax (***Fax Servers***):** solución ideal para organizaciones que tratan de reducir el uso del teléfono pero necesitan enviar documentos por fax.

- **Servidores** FTP **(***FTP Servers***):** Uno de los servicios más antiguos de Internet, File Transfer Protocol permite mover uno o más archivos.

- **Servidores Groupware (***Groupware Servers***):** software diseñado para permitir colaborar a usuarios, sin importar la localización, Internet, Intranet o virtual.

- **Servidores IRC (***IRC Servers***):** opción para usuarios que buscan la discusión en tiempo real, Internet Relay Chat, consiste en redes de servidores separadas que permiten que los usuarios conecten el uno al otro vía una red IRC.

- **Servidores de Listas (***List Servers***):** ofrecen una manera mejor de manejar listas de correo electrónico, bien sean discusiones interactivas abiertas al público o listas unidireccionales de anuncios, boletines de noticias o publicidad.

- **Servidores de Correo (***Mail Servers***):** mueven y almacenan correo electrónico a través de las redes corporativas (LANs WANs) e Internet.

- **Servidores de Noticias (***News Servers***):** actúan como fuente de distribución y entrega para los grupos de noticias públicos accesibles por la red de noticias.

- **Servidores Proxy (***Proxy Servers***):** se sitúan entre un programa del cliente (típicamente un navegador) y un servidor externo (típicamente otro servidor web) para filtrar peticiones, mejorar el funcionamiento y compartir conexiones.

- **Servidores Telnet (***Telnet Servers***):** permite a usuarios entrar en una maquina huésped y realizar tareas como si estuviera trabajando directamente en ésta.

- **Servidores Web (***Web Servers***):** sirve contenido estático a un navegador, carga un archivo y lo sirve a través de la red al cliente.

6.1.1 Características de Hardware

Hardware para Producción:

- ✓ Aplicación WEB.
  - Servidor:
    - Memoria RAM de 2 GB.
    - Procesador de 3 GB.
    - Disco Duro de 100 GB.
    - Tarjeta de Red.
  - Cliente:
    - Equipo de escritorio:
      - Memoria RAM 512 KB.
      - Disco Duro de 60 GB.
      - Tarjeta de Red o Modem.

Hardware para Desarrollo

- ✓ Aplicación WEB.
  - Servidor:
    - Memoria RAM de 2 GB.
    - Procesador de 2 GB.

- Disco Duro de 100 GB.
- Tarjeta de Red.

- Cliente:
  - Un equipo de escritorio (por programador):
    - Memoria RAM 512 KB.
    - Disco Duro de 60 GB.
    - Tarjeta de Red o Modem.

6.1.2 Características de Software

Software para Producción y Desarrollo:

- ✓ Aplicación WEB.
  - Cliente:
    - Navegador de Internet
  - Servidor:
    - JDK 5.0
    - Apache Tomcat 5.5
    - Base de Datos MySQL
    - Sistema Operativo Linux

6.2 Servidor de Aplicaciones

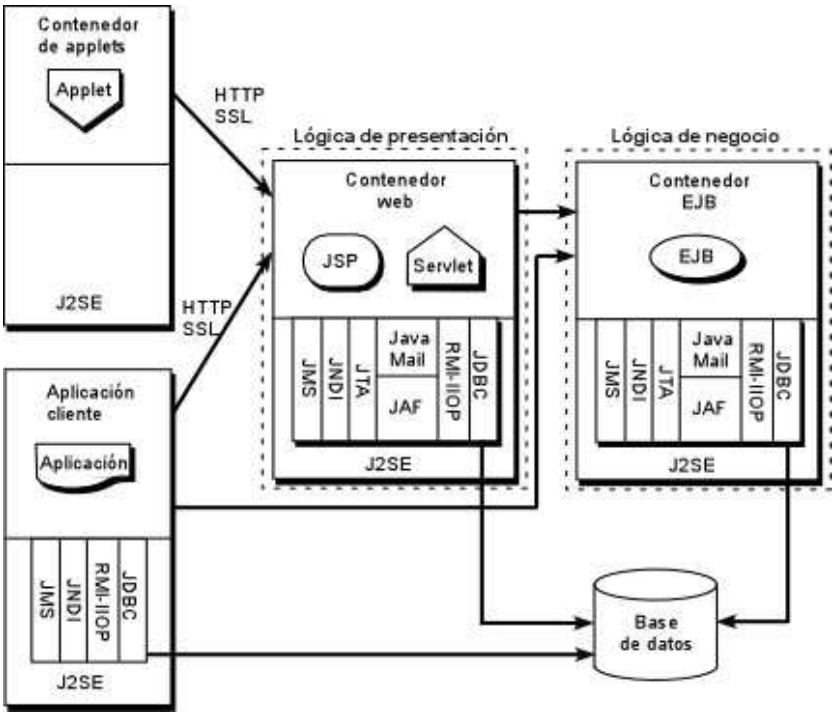
El término de servidor de aplicaciones se relaciona con el concepto de sistema distribuido, que permite mejorar tres aspectos fundamentales en una aplicación que son; la alta disponibilidad, la escalabilidad y el mantenimiento.

- **Alta disponibilidad:** se refiere a que un sistema debe funcionar las 24 horas del día los 365 días al año. Para alcanzar esta característica es necesario el uso de técnicas de balanceo de carga y de recuperación ante fallos.

- **Escalabilidad:** capacidad de hacer crecer un sistema cuando se incrementa la carga de trabajo (el número de peticiones). Cada máquina tiene una capacidad finita de recursos y por lo tanto sólo puede servir un número limitado de peticiones.

- **Mantenimiento:** se refiere a la versatilidad para actualizar, depurar fallos y mantener un sistema; una solución al mantenimiento es la construcción de la lógica de negocio en unidades reusables y modulares.

El estándar J2EE permite el desarrollo de aplicaciones empresariales de una manera sencilla y eficiente, una aplicación desarrollada con estas tecnologías permite ser desplegada en cualquier servidor de aplicaciones o servidor web que cumpla con el estándar. Un servidor de aplicaciones es una implementación de la especificación J2EE.



Arquitectura J2EE

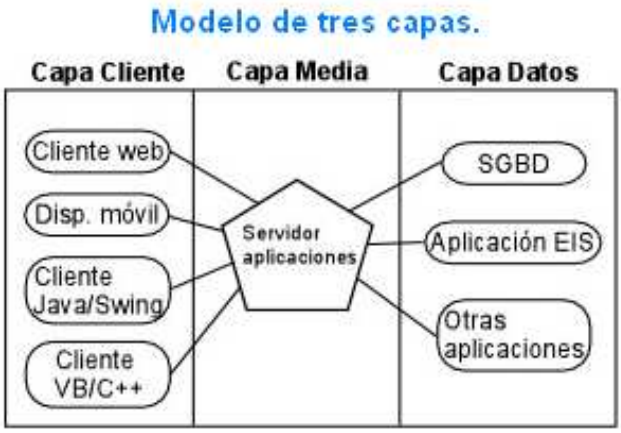
- **Cliente web** (contenedor de applets): Es un navegador e interactúa con el contenedor web haciendo uso de HTTP. Recibe páginas HTML o XML y puede ejecutar applets y código JavaScript.

- **Aplicación cliente:** Clientes que no se ejecutan en el navegador y utilizan cualquier tecnología para comunicarse con el contenedor web o con la BD.

- **Contenedor web:** Es el servidor web, la parte visible del servidor de aplicaciones. Utiliza protocolos HTTP y SSL para comunicarse.

- **Servidor de aplicaciones:** Proporciona servicios que soportan ejecución y disponibilidad de aplicaciones desplegadas, es lo más importante del sistema.

En comparación con una estructura tradicional de dos capas de un servidor web, un servidor de aplicaciones proporciona una estructura de tres capas, que permite estructurar el sistema de forma eficiente. No todas las aplicaciones empresariales necesitan un servidor de aplicaciones para funcionar, ya que una aplicación pequeña que acceda a una BD sencilla y no distribuida probablemente no necesite un servidor de aplicaciones, bastará solo con un servidor web (usando servlets y jsp).



Algunos de los servidores de aplicaciones más utilizados son los siguientes:

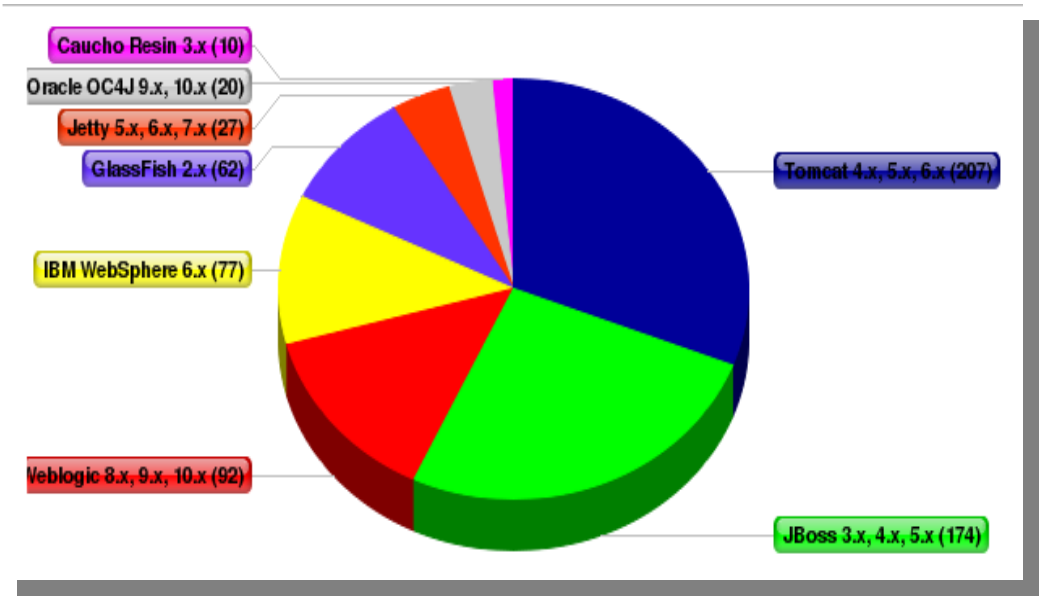
- Oracle WebLogic
- JBoss
- IBM WebSphere
- Sun-Netscape IPlanet
- Sun One
- Oracle IAS
- Borland AppServer

Un ejemplo común del uso de servidores de aplicación y de sus componentes son los portales de Internet, que permiten a las empresas la gestión y divulgación de su información, y un punto único de entrada a los usuarios internos y externos. Teniendo un servidor de aplicación, se puede tener acceso a información y servicios (Web) de manera segura y transparente, desde cualquier dispositivo.

6.2.1 Comparación de Servidores de Aplicaciones

Feature Description	Glassfish 2.1	JBoss 4.x	WebSphere 2.0	WebLogic 10
Java EE 5 compliance	Yes	Partial	Yes	Yes
JSP 2.1 and Servlet 2.5	Yes	Yes	Yes	Yes
EJB 3.0 capable	Yes	Yes	Yes	Yes
JavaServer Faces 1.2	Yes	Yes	Yes	Yes
Custom plug-in support	Yes	Yes	Yes	Yes
Business-rules engine support	Yes	Available	Available	Available
Hibernate 3.x support	Av ailable	Yes	Available	Available
JBoss Seam support	Available	Yes	Yes	Available
Clustering support	Yes	Yes	Yes	Yes
JAX-WS / JAX-B 2.x	Yes	Available	Yes	Yes
Eclipse IDE connector	Yes	Yes	Yes	Yes
Ease of operation rating 1 to 10	7	6	9	6

Servidores de Aplicaciones más utilizados.



Glassfish:

- Código abierto.
- Fácil instalación.
- Soporte completo con Java EE 5.
- Integración total con Netbeans.
- Basta documentación sobre uso, administración y desarrollo.
- Consola de administración amigable.
- Respaldo de Oracle.

JBoss:

- Código abierto.
- Fácil instalación.
- Para J2EE 5.
- El IDE soportado para JBoss es Eclipse.
- Mucha documentación sobre uso, administración, desarrollo.
- Respaldo de RedHat.
- JBoss Enterprise Middleware Services: JBoss Application Server, * JBoss Cache.
- Funciona con cualquier SGBD soportado por Hibernate.
- SSO/LDAP: usa soluciones de single sign on (SSO) de Tomcat y JBoss.
- Soporta Java Server Faces.
- Java Management Extensión
- Compatibilidad 100% con J2EE.
- Frameworks: puede utilizar Struts, Spring MVC, Sun JSF-RI, AJAX o MyFaces.

ORACLE Weblogic:

- Servidor de aplicaciones Java EE desarrollado por BEA Systems, posteriormente adquirido por Oracle, se ejecuta en Unix, Linux, Windows y otras plataformas.

- Puede utilizar Oracle, DB2, SQL Server y otras BD que usen JDBC.
- WebLogic Server es parte de Oracle WebLogic Platform que contiene:
  - Portal, que incluye servidor de comercio y servidor de personalización Weblogic Integration,
  - Weblogic Workshop, una IDE para Java
  - JRockit, una máquina virtual Java (JVM) para CPUs de Intel
  - Mensajería nativa JMS
  - J2EE Connector Architecture
  - Conector WebLogic/Tuxedo
  - Conectividad COM+
  - Conectividad CORBA
  - Conectividad IBM WebSphere MQ

Websphere: IBM WebSphere Application Server (WAS, servidor de aplicaciones WebSphere)

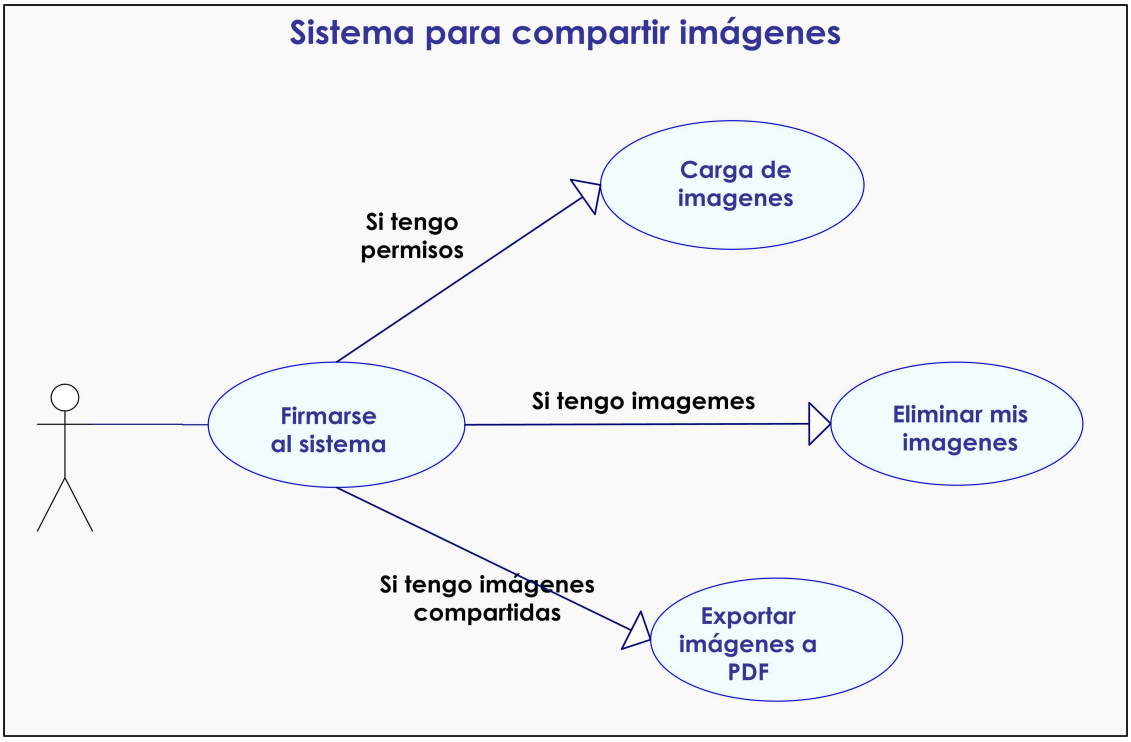
- Producto estrella dentro de la familia WebSphere de IBM.
- Construido usando estándares abiertos como J2EE, XML y Servicios Web.
- Funciona con varios servidores web (Apache HTTP Server, Netscape Server, Microsoft IIS, IBM HTTP Server para i5/OS, IBM HTTP Server para z/OS, y el sistema operativo AIX/Linux/Microsoft Windows/Solaris.
- Cumple J2EE 1.5
- Gestión sencilla.
- Facilita la administración de diversas topologías WAS edición.
- Aplicación a nivel de negocio
- Configuración basada en propiedades
- Simplifica la gestión de administración automática: un administrador puede actualizar la configuración de un WAS 7 utilizando un archivo de configuración.

Tomcat 7x:

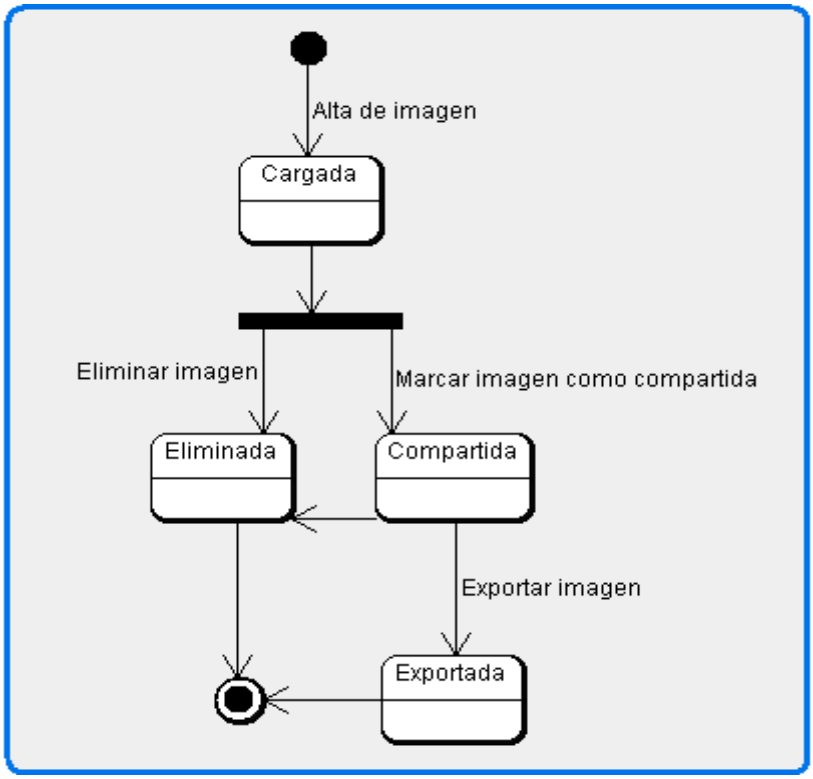
- Funciones básicas con HTTP.
- Contenedor de servlets rediseñado.
- Motor JSP rediseñado con Jasper.
- Java Management Extensions (JMX), JSP Y administración basada en Struts
- Recolección de basura reducida.
- Capa envolvente para Windows y Unix para la integración de las plataformas.
- Análisis rápido JSP.
- Soporte para Unified Expression Language 2.1.
- Diseñado para funcionar en Java SE 5.0 y posteriores.
- Implementado de Servlet 3.0 JSP 2.2 y EL 2.2.
- Mejoras para detectar y prevenir "fugas de memoria" en las aplicaciones web.
- Limpieza interna de código.
- Soporte para inclusión de contenidos externos directos en una aplicación web.

7. ANEXOS.

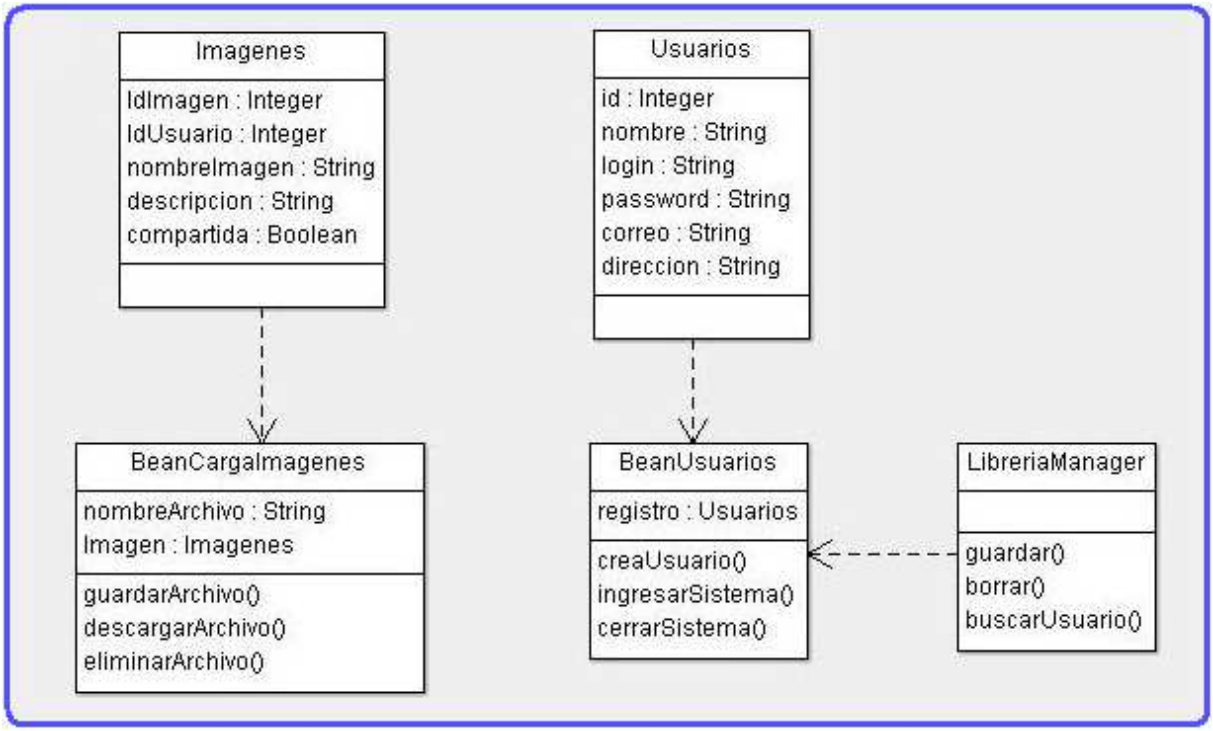
- **1. DIAGRAMA DE CASO DE USO**



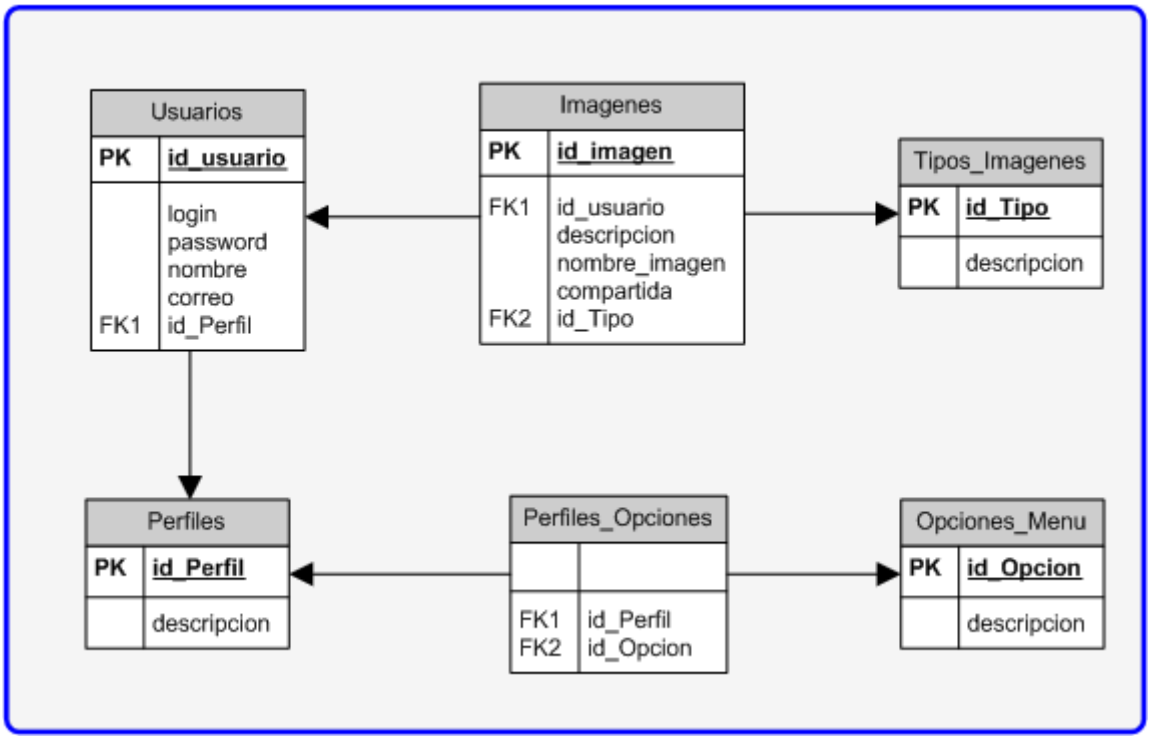
- **2. DIAGRAMA DE ESTADOS**



- **3. DIAGRAMA DE CLASES.**



- **4. DIAGRAMA ENTIDAD-RELACIÓN**



8. CONCLUSIÓN.

A lo largo de este trabajo se han expuesto cada una de las etapas a considerar para llevar a cabo el desarrollo de un sistema, desde el cómo elegir un ciclo de vida acorde a al tamaño, hasta el tipo de pruebas que se pueden realizar para garantizar en cierto punto la calidad del producto final; cada una de las etapas descritas juegan un papel importante en el proceso, pero lo más importante y lo que determina el tiempo y esfuerzo a invertir, es conocer el Análisis puesto que se debe conocer a detalle cuáles son los requerimientos del usuario para tener bien definido desde un inicio qué es lo que se debe desarrollar antes de realizar cualquier inversión de recursos, y retrabajar en cada fase, corrigiendo errores de lógica y funcionamiento.

Una parte importe del Análisis, radica en la correcta definición del alcance del proyecto, así como acotarlo en cuanto a los entregables finales del sistema, ya que si a lo largo del proceso de implantación llega a ocurrir un cambio no previsto dentro del alcance previamente definido, éste deberá tomarse como un control de cambio que se realizara en una segunda etapa del proyecto pero nunca como parte del proyecto original. Después del Análisis es importante apoyarse con el equipo de desarrollo para poder definir un plan de trabajo con tiempos más cercanos a los reales para la duración del proyecto.

Gracias a un buen Análisis y Diseño se logrará la elaboración de la documentación necesaria, por ejemplo diagramas de estado, diagramas de secuencia, etc., los cuales nos llevaran a no dejar a la imaginación del desarrollador el qué debe hacer y cómo debe de funcionar la aplicación y con esto asegurar que el producto final entregado sea lo que el usuario solicito o lo más cercano a ello.

En realidad no existe receta mágica para que nuestro desarrollo resulte perfecto y se entregue de acuerdo a los tiempos planteados y con el total de especificaciones requeridas, hay que tomar en cuenta que existen circunstancias externas que pueden impactar nuestra entrega, desde los conocimientos que los desarrolladores tengan para plantear los diseño en el código, hasta que se hagan las pruebas unitarias y de integración adecuadas, pues en ocasiones sucede que los desarrolladores realizan prueban aisladas de cada componente y funcionan adecuadamente, pero a la hora de conjuntar los módulos, estos fallan; lo que probablemente sea resultado de la falta de comunicación e integración entre el equipo de desarrollo para llevar a cabo las pruebas del sistema.

Otro punto importante a considerar dentro del desarrollo de nuestro sistema, es la usabilidad, que como ya se vio, se refiere al grado de facilidad con que el usuario puede realizar ciertas tareas dentro del sistema; es decir que aparte de que el producto entregado realice las funciones que debe tal cual los requerimientos, también debe ser amigable y practico al utilizarse, ya que si su uso resulta difícil, tedioso o de plano no llama la atención, de nada habrá servido haber construido el mejor sistema si no resulta atractivo para el usuario final, su empleo debe resultar interactivo, dinámico, fácil, intuitivo y cómodo, centrándonos siempre en un diseño orientado al usuario y obviamente sin dejar del lado el funcionamiento para así poder entregar un sistema completo que integre la usabilidad con la funcionalidad y que al final nos llevará a lograr un mayor éxito en el desarrollo de aplicaciones.

Por lo tanto, será de vital importancia involucrar al usuario final durante todo el proceso de desarrollo del sistema, desde la validación de los documentos de análisis y diseño, pasando por verificación de los prototipos de pantallas y sobre todo en las pruebas integrales del sistema. Es por ello que para poder llevar a cabo un buen proceso de desarrollo de aplicaciones se deben tomar en cuenta los actores involucrados como los desarrolladores, el usuario, líderes de proyecto, analistas, entre otros, siendo igual de importante llevar a cabo todos los pasos que hasta ahora hemos revisado en el presente trabajo, así como el saber y poder trabajar en equipo.

9. REFERENCIAS.

LIBROS

I. JAMES RUMBAUGH, IVAR JACOBSON, GRADY BOOCH **"El Lenguaje Unificado de Modelado Manual de Referencia"**, Madrid, España: Pearson Education, Primera edición, 2000.

II. ROGER S. PRESSMAN **"Ingeniería del software Un enfoque Práctico"**, Madrid, España: McGraw-Hill, Quinta edición, 2002.

III. JOSEPH SCHMULLER **"Aprendiendo UML en 24 Horas"**, México: Pearson Education, Primera edición, 2000.

IV. CRAIG LARMAN, **"UML y Patrones Introducción al Análisis y Diseño Orientado a Objetos"** México: Prentice-Hall, 2003.

V. HERBERT SCHILDT, **"Fundamentos de Programación en Java 2"** Colombia, Bogotá DC.: McGraw-Hill, Primera edición, 2002.

VI. KATHY SIERRA y BERT BATES, **"SCJP Sun Certified Programmer for Java 6",** McGraw-Hill Osborne Media; 1 edition (June 24, 2008).

VII. Pertenecientes a la colección : "Aprenda …, como si estuviera en primero":

JAVIER GARCÍA DE JALÓN, JOSÉ IGNACIO RODRÍGUEZ, AITOR IMAZ, **"Aprenda Java como si estuviera en primero"**, España, Navarra, Tecnun San Sebastián, Enero 2000.

JAVIER GARCÍA DE JALÓN, JOSÉ IGNACIO RODRÍGUEZ, AITOR IMAZ, **"Aprenda Servlets de Java como si estuviera en segundo"**, España, Navarra Tecnun, San Sebastián, Abril 1999.

INTERNET

- http://www.programacionenjava.com/blog/tag/tutorial/
- http://www.programacion.com/articulo/internacionalizacion_de_programas_java_140/10
- http://www.finderit.com
- http://www.taringa.net/posts/ebooks-tutoriales/2011282/Libros-y-Apuntes-Materia-Analisis-de-Sistemas-UTN-(ASI).html
- http://es.wikipedia.org/wiki/Pruebas_de_software
- http://msdn.microsoft.com/eses/library/bb972232.aspx#mainSection#mainSection
- http://www.arrakis.es/~abelp/ApuntesJava/ClasesIV.htm#La referencia this
- http://pegasus.javeriana.edu.co/~fwj2ee/descargas/metodologia(v1.0).pdf