Resumen de IPOO

Unidad 1 - El Proceso de Desarrollo de Software

Los sistemas de software actuales suelen resolver problemas complejos que requieren soluciones confiables, eficientes y capaces de adaptarse dinámicamente a cambios en las necesidades de los clientes o usuarios. El desarrollo de un sistema de software de estas características es un proceso que tiene un ciclo de vida conformado por etapas que pueden organizarse de diferentes formas. El producto final de este proceso es un sistema de software.

El proceso se realiza en el marco de un proyecto que establece un cronograma y un presupuesto para la elaboración de un producto. El éxito del proyecto está ligado a la calidad del producto, pero también es fundamental que se complete con los costos previstos en el presupuesto y los tiempos establecidos en el cronograma.

Aunque el desarrollo de software es una actividad creativa, requiere de la aplicación de un paradigma que guíe, oriente y sistematice cada etapa. Un paradigma de programación brinda:

- **Un principio** que describe propiedades generales que se aplican a todo el proceso de desarrollo.
- **Una metodología** que consta de un conjunto integrado de métodos, estrategias y técnicas que aseguran la aplicación del principio.
- **Un conjunto de herramientas** que soportan y facilitan la aplicación de la metodología.

Dos de las herramientas fundamentales del proceso de desarrollo de software, son el lenguaje de modelado y el lenguaje de programación.

Ciclo de vida

El ciclo de vida de un sistema de software comienza cuando se identifica el problema o se concibe la idea que le da origen y termina cuando el producto deja de utilizarse. Un modelo de ciclo de vida propone un enfoque específico para establecer y ordenar las etapas que conforman el proceso de desarrollo de un producto de software.

En los modelos de ciclo de vida secuenciales las etapas se ordenan en forma lineal. El resultado de una etapa es el insumo para la siguiente hasta alcanzar el producto final. En los modelos de ciclo de vida evolutivos se desarrolla una versión inicial del producto que luego se va refinando al incorporar nuevas funcionalidades o al mejorar los atributos de calidad. El proceso también se organiza en etapas, pero pueden solaparse en el tiempo. En los modelos evolutivos las primeras versiones se entregan sin conocer los alcances del producto final.

La documentación es una etapa transversal a todo el proceso. En cada etapa se produce un documento que sirve de insumo para las que siguen. El software está formado por el código escrito en un lenguaje de programación y el conjunto de documentos que se generan durante el proceso de desarrollo.

Con diferentes nombres dependiendo del modelo de ciclo de vida, las etapas del proceso de desarrollo de software son:

★ **Desarrollo de los Requerimientos:** Un proyecto será exitoso si el sistema es una solución para el problema, para ello deben definirse con precisión los requerimientos.

El resultado de esta etapa específica:

- ➔ Qué problema tiene que ser resuelto.
- ➔ Por qué es un problema y por lo tanto requiere solución.
- ➔ Qué cualidades debe exhibir la solución, qué funcionalidades debe brindar y qué restricciones debe cumplir.
- ➔ Quiénes tienen la responsabilidad de participar en la construcción de la solución.

★ **Diseño:** A partir de los requerimientos se diseña una solución para el problema especificado. El resultado de esta etapa es una especificación de los módulos que integrarán el sistema y el modo en que se relacionan entre sí. La especificación puede establecer también casos de prueba o tests que se aplicarán en la verificación.

★ **Implementación:** A partir de la especificación producida durante la etapa de diseño, los desarrolladores o programadores generan el programa escrito en un lenguaje de programación y toda la documentación referida al código. Es importante que el programa implementado mantenga la estructura de la solución especificada en la etapa de diseño. Cada módulo de diseño debería corresponderse con una unidad de código implementada, con cierta independencia del sistema completo**.**

★ **Verificación y depuración:** La verificación evalúa si el sistema satisface la especificación de requerimientos. Durante la depuración se corrigen los errores que se detectan. Cuando el sistema está dividido en módulos, cada unidad de código se verifica y depura por separado y luego se verifica y depura la integración de los módulos. Es importante que al menos una parte de la verificación la lleven a cabo personas ajenas a la implementación. Los casos de prueba pueden haberse establecido en el diseño o pueden ser definidos en esta misma etapa. El término validación se utiliza para referirse a la evaluación de un sistema respecto a las necesidades reales del usuario, que en ocasiones no corresponden a los requerimientos especificados.

El lenguaje de modelado comienza a utilizarse durante el desarrollo de requerimientos y permite elaborar diferentes tipos de diagramas. El lenguaje de programación afecta fundamentalmente a la etapa de implementación.

Calidad

En un sentido estricto la calidad de un sistema de software se evalúa considerando el nivel de satisfacción que alcanza el usuario o cliente a partir del momento que comienza a utilizarlo. Un mecanismo menos exigente evalúa la calidad del sistema con relación a los requerimientos acordados.

En cualquier caso la calidad se evalúa a partir de diferentes atributos o factores, entre ellos:

- ➢ **Correctitud:** Un producto de software es correcto si actúa de acuerdo a los requerimientos especificados.
- ➢ **Eficiencia:** Un producto de software es eficiente si tiene una baja demanda de recursos de hardware, en particular tiempo de CPU, espacio de memoria y ancho de banda.
- ➢ **Portabilidad:** Un producto de software es portable si puede ejecutarse sobre diferentes plataformas de hardware y de software.
- ➢ **Simplicidad:** Un producto de software es simple si es fácil de usar, su interfaz es amigable y no requiere demasiado entrenamiento ni capacitación por parte del usuario.

- ➢ **Robustez:** Un producto de software es robusto si reacciona adecuadamente aun en circunstancias imprevisibles.
- ➢ **Disponibilidad:** Un producto de software tiene buena disponibilidad si puede ser usado en el momento que el usuario lo necesita y el rendimiento está dentro de parámetros establecidos.
- ➢ **Legibilidad:** Un producto de software es legible si un desarrollador o incluso otros miembros del equipo de desarrollo, puede leerlo e interpretar su estructura y contenido fácilmente.

Productividad

La productividad está ligada al costo y al tiempo que demanda la concepción y construcción de un sistema e incide en cada etapa de su desarrollo.

La productividad está vinculada a dos factores fundamentales, extensibilidad y reusabilidad.

Extensibilidad: Un producto de software es extensible si el costo y el tiempo que demanda un cambio en la especificación, es consistente con la envergadura del cambio.

Reusabilidad: Un módulo de software o una colección de módulos es reusable sí puede utilizarse para la construcción de diferentes aplicaciones.

La extensibilidad afecta fundamentalmente al mantenimiento, pero depende de las etapas anteriores. Los cambios en un sistema con un diseño flexible tendrán un costo menor que los que provocará la extensión de una solución con un diseño pobre.

La reusabilidad afecta a todas las etapas. Cuando el equipo de desarrollo resuelve un problema puede reusar requerimientos, módulos de diseño o código de sistemas anteriores.

Ambas cualidades están ligadas a la legibilidad, un factor de calidad que el usuario no percibe, pero que sí es valioso para el equipo de desarrollo. La documentación y estructuración del código favorecen la extensión y el reuso.

Programación orientada a objetos

El principio fundamental del paradigma de programación orientada a objetos es desarrollar un sistema de software en base a las entidades relevantes del problema que le da origen.

Una metodología orientada a objetos parte del reconocimiento de los objetos del problema. Cada objeto del problema es una entidad que puede caracterizarse a través de sus atributos y su comportamiento. Los atributos y el comportamiento permiten agrupar a los objetos en clases. Una metodología orientada a objetos abarca el ciclo de vida completo de un producto de software y brinda herramientas adecuadas para cada etapa del proceso.

El conjunto de herramientas incluye un lenguaje de modelado y un lenguaje de programación. Un lenguaje de modelado es una notación artificial que permite describir la estructura y las componentes de un sistema de software. Un lenguaje de modelado consistente con la programación orientada a objetos, permite representar a la colección de clases y sus relaciones a través de un diagrama de clases. Un lenguaje de programación orientado a objetos brinda mecanismos que facilitan la implementación del sistema diseñado.

Una clase define los atributos que caracterizan a un conjunto de objetos. Las clases se vinculan entre sí a través de diferentes mecanismos de relación. El documento producido en esta etapa incluye un diagrama de clases que modela la colección de clases y sus relaciones. En la etapa de diseño se establece el comportamiento de los objetos y se completa la especificación de las clases y relaciones.

Unidad 2 - Objetos y Clases

El concepto de objeto

Durante el desarrollo de requerimientos de un sistema de software el analista es responsable de identificar los objetos del problema y caracterizarlos a través de sus atributos. En la etapa de diseño se completa la representación modelando también el comportamiento de los objetos.

Un **objeto del problema** es una entidad, física o conceptual, caracterizada a través de atributos y comportamiento. El comportamiento queda determinado por un conjunto de servicios que el objeto puede brindar y un conjunto de responsabilidades que debe asumir. las entidades identificadas durante el desarrollo de requerimientos o el diseño.

Cuando el sistema de software está en ejecución, se crean objetos de software.

Un **objeto de software** es un modelo, una representación de un objeto del problema. Un objeto de software tiene una identidad y un estado interno y recibe mensajes a los que responde ejecutando un servicio. El estado interno mantiene los valores de los atributos. Los objetos de software se comunican a través de mensajes para solicitar y brindar servicios. La ejecución de un servicio puede modificar el estado interno del objeto que recibió el mensaje y/o computar un valor. Cada representación que modela en ejecución a una entidad del problema.

El **modelo computacional** propuesto por la programación orientada a objetos es un mundo poblado de objetos comunicándose a través de mensajes. La ejecución se inicia cuando un objeto recibe un mensaje y en respuesta a él envía mensajes a otros objetos.

El concepto de clase

Durante el desarrollo de requerimientos de un sistema de software se identifican los objetos relevantes del problema, se los caracteriza a través de sus atributos y se los agrupa en clases**.**

Desde el punto de vista del diseño de un sistema de software, una clase es un patrón que establece los atributos y el comportamiento de un conjunto de objetos.

Durante la implementación, el programador escribe el código de cada clase en un lenguaje de programación.

En la implementación de un sistema de software una clase es un módulo de software que puede construirse, verificarse y depurarse con cierta independencia a los demás.

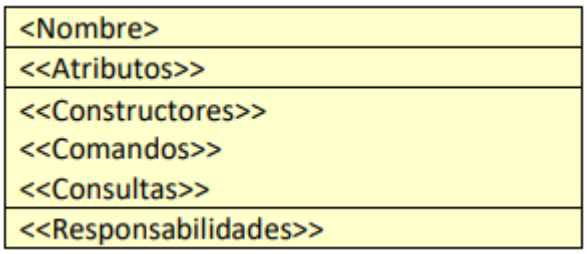
Desde el punto de vista estático un sistema de software orientado a objetos es una colección de clases relacionadas. Desde el punto de vista dinámico un sistema de software orientado a objetos es un conjunto de objetos comunicándose.

El diseño de una clase

Durante el desarrollo de requerimientos el analista construye un diagrama de clases que modela el problema. El diseñador del sistema completa el diagrama de clases para modelar la solución.

Un **diagrama de clases** es una representación visual que permite modelar la estructura de un sistema de software a partir de las clases que lo componen.

El diagrama de clases se construye usando un lenguaje de modelado. El diagrama de cada clase en el lenguaje de modelado tiene la siguiente forma:



El **nombre** de una clase representa la abstracción del conjunto de instancias. Por lo general es un sustantivo y debe elegirse cuidadosamente para representar al conjunto completo.

Un **atributo** es una propiedad o cualidad relevante que caracteriza a todos los objetos de una clase. Es posible distinguir los **atributos de clase** de los **atributos de instancia**. En el primer caso, el valor es compartido por todos los objetos que son instancias de la clase. Los valores de los atributos de instancia varían en cada objeto.

Un **servicio** es una operación que todas las instancias de una clase pueden realizar. Los servicios pueden ser **métodos** o **constructores**.

Un **método** es un servicio que un objeto ejecuta en respuesta a un mensaje.

Un **constructor** es un servicio que se invoca para crear un objeto.

Los métodos pueden ser **comandos** o **consultas.**

Un **comando** es un método que al ejecutarse modifica el valor de uno o más atributos del objeto que recibió el mensaje.

Una **consulta** es un método que al ejecutarse no modifica el valor de ningún atributo del objeto que recibió el mensaje.

Cada comando o consulta tiene una **funcionalidad** que establece su propósito

Una **responsabilidad** representa un compromiso o un requisito para una clase.

El conjunto de servicios y las responsabilidades determinan el comportamiento de las instancias de una clase.

La implementación de una clase

La implementación requiere elegir un lenguaje de programación. Un lenguaje orientado a objetos permite retener durante la implementación la organización de las clases modeladas durante el desarrollo de requerimientos y la etapa de diseño.

La palabra reservada class está seguida por un identificador que es el nombre de la clase. Java es sensible a las minúsculas y mayúsculas. Las { } son los delimitadores de cada unidad de

código. Existen otros delimitadores como los corchetes y los paréntesis. El símbolo // precede a un comentario de una línea. Los símbolos /* */ delimitan a un comentario de varias líneas. El símbolo ; termina cada instrucción.

Los miembros de una clase son atributos y servicios. Manteniendo la misma clasificación que la propuesta por el lenguaje de modelado, los atributos pueden ser de clase o de instancia. Los servicios pueden ser constructores, comandos o consultas. En todos los casos están formados por un encabezamiento y un bloque de instrucciones delimitado por llaves.

Los atributos se definen a través de la declaración de variables. Un atributo de clase se declara como una variable estática. En la declaración de una variable, el tipo precede al nombre de la variable. Un tipo elemental determina un conjunto de valores y un conjunto de operaciones que se aplican sobre estos valores. En ejecución, una variable de un tipo elemental mantiene un valor que corresponde al tipo y participa en las operaciones establecidas por su tipo. Java brinda siete tipos de datos elementales. Una variable de un tipo elemental se utiliza como operando en cualquier expresión en la que se apliquen operadores con los cuales su tipo sea compatible. El mecanismo de conversión implícito entre tipos elementales permite escribir expresiones mixtas, esto es, con operandos de distinto tipo.

Un **constructor** es un servicio provisto por la clase y se caracteriza porque recibe el mismo nombre que la clase. El constructor se invoca cuando se crea un objeto y habitualmente se usa para inicializar los valores de los atributos de instancia. En Java, si una clase no define explícitamente un constructor, el compilador crea automáticamente uno, en ese caso los atributos son inicializados por omisión. Una clase puede brindar varios constructores, siempre que tengan diferente número o tipo de parámetros. Si la clase incluye uno o más constructores, el compilador no agrega ningún otro.

Los comandos son servicios que modifican los valores de uno o más de los atributos del objeto que recibe el mensaje. Las consultas son servicios que no modifican los valores de los atributos del objeto que recibe el mensaje y por lo general retornan un valor. Un comando puede retornar también un valor. Los comandos y consultas conforman los métodos de una clase. Cada método está precedido por el tipo del resultado. Si el nombre de un método está precedido por un tipo elemental o una clase, el bloque debe incluir una instrucción de retorno. La instrucción de retorno comienza con la palabra return y sigue con una expresión que debe ser compatible con el tipo del resultado. Si un método no retorna un resultado su nombre va precedido por la palabra void.

Edición, compilación y ejecución

La edición del código de cada clase se realiza dentro de un entorno de desarrollo que integra también recursos para compilar, depurar, ejecutar código y crear aplicaciones autónomas. La sigla IDE se utiliza justamente para referirse a un entorno integrado de desarrollo.

El desarrollador edita el código fuente de cada clase y luego las compila. Una de las clases debe incluir un método con la siguiente signatura:

public static void main (String []a)

En Java la ejecución comienza cuando se invoca el método **main** de una clase específica. Aunque Java es un lenguaje que admite la programación orientada a objetos, incluye varias

características que lo alejan de este modelo de programación. En particular la clase que define al método main no modela a objetos del problema. En Java una clase puede ser utilizada para modelar objetos de un problema, tal como propone la programación orientada a objetos, pero también puede ser utilizada con otro propósito. Java no se considera un lenguaje orientado a objetos puro porque admite otros modelos de programación. Todas las clases que conforman el sistema deben haberse compilado antes de que se invoque a main. La ejecución provocará la creación de algunos objetos de otras clases que recibirán mensajes y probablemente crearán objetos de otras clases.

La verificación y depuración de una clase

Un sistema de software orientado a objetos está conformado por una colección de clases. Cada clase implementada debe ser verificada individualmente antes de integrarse con el resto de las clases que conforman la colección. La verificación de una clase aspira a detectar y depurar errores de ejecución o de aplicación. Los errores de compilación ya han sido corregidos antes de comenzar la verificación.

Los **errores de ejecución** provocan la terminación anormal del sistema. Para detectarlos es necesario anticipar distintos flujos de ejecución, definiendo casos de prueba que conducen a una terminación anormal. Java previene muchos errores de ejecución imponiendo chequeos durante la compilación, sin embargo no controla situaciones como por ejemplo división por 0.

Los **errores de aplicación**, también llamados errores lógicos, son probablemente los más difíciles de detectar, el sistema no termina en forma anormal, pero los resultados no son correctos. La causa puede ser que no se hayan especificado correctamente los requerimientos o la funcionalidad, o bien que el diseñador haya elaborado correctamente la especificación, pero la implementación no sea consistente con ella.

En un sistema de mediana o alta complejidad probablemente los roles de analista, diseñador, desarrollador y responsable de verificación y depuración sean ocupados por diferentes miembros del equipo de desarrollo. Sin embargo, con frecuencia el diseñador define un conjunto de casos de prueba que el responsable de la verificación debe utilizar. En todos los casos el objetivo es detectar y depurar los errores, considerando las responsabilidades asignadas a la clase y la funcionalidad establecida para cada servicio. Es importante considerar que la verificación muestra la presencia de errores, no garantiza la ausencia.

Una clase tester es aquella que verifica los servicios de una o más clases para un conjunto de casos de prueba. Los casos de prueba propuestos aspiran detectar errores internos en el código, no fallas externas al sistema o situaciones que no fueron previstas en el diseño.

Objetos y referencias

En Java los objetos son referenciados a través de variables. La ejecución de la instrucción:

PresionArterial mDia = new PresionArterial (115,60);

tiene el siguiente significado:

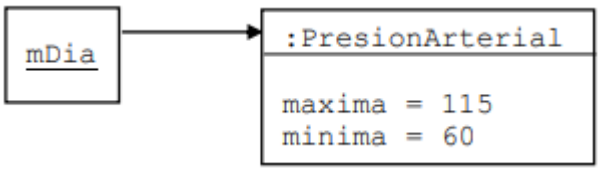
- ★ **Declara** la variable mDia.
- ★ **Crea** un objeto de clase PresionArterial
- ★ **Invoca** al constructor de la clase PresionArterial

★ **Liga** el objeto a la variable mDia

Una **variable ligada** mantiene una **referencia** al **estado interno** de un objeto.

Un **diagrama de objetos** es una representación visual que modela el estado interno de uno o más objetos en ejecución.

El diagrama de objetos, luego de ejecutada la asignación anterior es:



Clases y tipos de datos

Cuando el analista o el diseñador de un sistema orientado a objetos especifica una clase, establece sus atributos y servicios. Los servicios incluyen constructores y métodos, estos últimos conformados por comandos y consultas. En la implementación, una clase con esta estructura define un **tipo de dato** a partir del cual se pueden **declarar variables**.

Un **tipo de datos** establece un conjunto de valores y un conjunto de operaciones que se aplican sobre estos valores.

Cuando una clase define un tipo de datos, el conjunto de valores queda determinado por los valores de los atributos, el conjunto de operaciones lo definen los servicios provistos por la clase. Una variable declarada de un tipo definido por una clase, no mantiene un valor dentro del tipo, sino una referencia a un objeto cuyo estado interno mantiene un valor del tipo.

Clases y paquetes

Un **paquete** en Java es un contenedor que agrupa un conjunto de clases con características comunes. El uso de paquetes favorece en primer lugar la reusabilidad porque permite utilizar clases que ya han sido diseñadas, implementadas y verificadas, fuera del contexto del sistema que se está desarrollando. También favorece la eficiencia porque el lenguaje brinda un repertorio reducido de recursos, pero permite agregar otros importando paquetes.

Variables y alcance

La declaración de una variable establece su **nombre**, su **tipo** y su **alcance**. El tipo puede ser elemental o una clase. El alcance de una variable determina su **visibilidad**, es decir, el segmento del código en el cual puede ser **usada**.

En Java una variable declarada en una clase, ya sea como atributo de clase o de instancia, es visible en toda la clase. Si se declara como privada, solo puede ser usada dentro de la clase, esto es, el alcance es la clase completa.

En Java una variable declarada local a un bloque, se crea en el momento que se ejecuta la instrucción de declaración y se destruye cuando termina el bloque que corresponde a la declaración. El alcance de una variable local es entonces el bloque en el que se declara, de modo que sólo es visible en ese bloque.

Una variable declarada como parámetro formal de un servicio se trata como una variable local que se crea en el momento que comienza la ejecución del servicio y se destruye cuando termina. Se inicializa con el valor del argumento o parámetro real. El pasaje de parámetros en Java es entonces por valor.

Mensajes y métodos

Cuando un objeto recibe un mensaje, su clase determina el método que se va a ejecutar en respuesta a ese mensaje. Cuando un objeto recibe un mensaje, el **flujo de control** se interrumpe y el control pasa al método que se liga al mensaje. Al terminar la ejecución del método, el control vuelve a la instrucción que contiene al mensaje. Java permite modelar también procesamiento paralelo, esto es, ejecutar varios métodos simultáneamente.

Si un método retorna un valor, el tipo de la expresión que sigue a la palabra a *return* debe ser compatible con el tipo que precede al nombre del método en el encabezamiento.

Si los valores los ingresa el usuario, es necesario **validar la entrada**.

El método *toString()* retorna la concatenación de los valores de los atributos del objeto que recibe el mensaje.

Parámetros y resultados de tipo clase

Un método puede recibir como parámetro o retornar como resultado a un objeto. En ambos casos, la variable que se recibe como parámetro o se retorna como resultado es de tipo clase.

Cuando en ejecución un mensaje se vincula a un método, el número de parámetros formales y reales es el mismo y los tipos son consistentes. Como se mencionó antes, en Java el pasaje de parámetros es por valor. Por cada parámetro formal se crea una variable y se le asigna el valor del parámetro real. Si el parámetro es de tipo clase, se asigna una referencia, de modo que el parámetro formal y el parámetro real mantienen una referencia a un mismo objeto.

Si un método modifica el valor de un parámetro formal, cuando la ejecución de ese método termina el cambio no persiste. En cambio, si un método modifica el estado interno de un objeto referenciado por un parámetro formal, cuando el método termina el estado interno se ha modificado.

La programación estructura recomienda que cada instrucción tenga un único punto de entrada y uno de salida, en particular un método debe incluir una única instrucción de retorno al final, o dos instrucciones, sólo en el caso de que el código contenga una única instrucción *if-else* con una instrucción simple en cada alternativa.

El método *toString()* retorna un resultado de tipo *String*, esto es, una cadena de caracteres. Expresado con mayor precisión, el método *toString()* retorna una referencia a un objeto de clase *String*. El mensaje *System.out.println* requiere como parámetro un objeto de clase *String*. Cuando un parámetro es de tipo clase, el método recibe una referencia.

Identidad, igualdad y equivalencia

Cada objeto de software tiene una **identidad**, una propiedad que lo distingue de los demás. La **referencia** a un objeto puede ser usada como propiedad para identificarlo.

Si varias variables están ligadas a un mismo objeto, todas mantienen una misma referencia, esto es, comparten una misma identidad.

Los **operadores relacionales** comparan los valores de las variables. Cuando se aplica el operador de igualdad sobre variables de tipo clase, se comparan los valores de las variables. El valor de cada variable es una referencia nula o ligada a un objeto.

Para comparar el **estado interno** de los objetos, se comparan los valores de los atributos de instancia.

Cuando una clase define un tipo de dato, por lo general va a incluir un método que decide si dos objetos que son instancias de esa clase, tienen el mismo estado interno. Por convención se utiliza el nombre *equals* para este método.

También es habitual que una clase incluya métodos para copiar y clonar. El primero copia el estado interno de un objeto en otro. El segundo crea un clon del objeto que recibe el mensaje; el nuevo objeto puede evolucionar luego de manera autónoma al original. Se le llama *copy* y *clone* a estos métodos.

El método *equals* por convención compara el estado interno del objeto que recibe el mensaje, con el estado interno del objeto que pasa como parámetro.

El estado del objeto que recibe el mensaje se accede directamente a través de los atributos de instancia. El estado del objeto que pasa como parámetro se accede indirectamente a través de las operaciones provistas por la clase.

La consulta requiere que el parámetro esté ligado.

Un diseño alternativo podría indicar expresamente que el método *equals* tiene la responsabilidad de controlar si el parámetro está ligado.

El método *equals* es una **operación binaria**, un operando es el objeto que recibe el mensaje, el otro operando es el objeto que pasa como parámetro.

El operador relacional = = decide si dos objetos tienen la misma identidad. El servicio *equals* decide si dos objetos son equivalentes.

El método *copy* modifica el estado interno del objeto que recibe el mensaje, con el estado interno del objeto que pasa como parámetro. El estado del objeto que recibe el mensaje se accede directamente a través de los atributos de instancia.

La consulta *clone* retorna la referencia a un objeto con el mismo estado interno que el objeto que recibe el mensaje.

Representación en memoria

La memoria de una computadora es el dispositivo en el que se almacenan datos e instrucciones. Toda la información se almacena utilizando el sistema de numeración binario. La estructura de la memoria es así sumamente simple y puede graficarse como:

Dirección	*Contenido*
00000000	11011111
00000001	10101010
00000010	11110000
00000011	00000001

Cada celda de memoria tiene una **dirección** y un **contenido.** Un conjunto de celdas consecutivas pueden agruparse para definir un **bloque de memoria** que contenga a una **unidad de información**, por ejemplo, una cadena de caracteres. El contenido de una celda puede ser una dirección en memoria, en ese caso la celda contiene una referencia a otro bloque de memoria. En el diagrama de memoria propuesto, el contenido de la cuarta celda es la dirección de memoria de la segunda celda.

En Java el tipo de una variable puede ser elemental o una clase. La **representación interna en memoria** es diferente en cada caso. Una variable de tipo elemental almacena un valor de su tipo. Una variable de tipo clase almacena una **referencia** a un objeto de software de su clase.

Cuando en ejecución se alcanza una declaración de una variable local de tipo elemental se reserva un **bloque de memoria.**

Cuando en ejecución se alcanza una **declaración de variable** cuyo tipo es una clase, la variable es de **tipo clase,** se reserva también un **bloque de memoria** que mantendrá inicialmente una **referencia no ligada.**

La **creación de un objeto** reserva un nuevo bloque de memoria para mantener el estado interno del objeto. El valor de la variable no pertenece al conjunto de valores que determina el tipo, sino que es una **dirección al bloque de memoria** en el que se almacena el estado interno del objeto.

El diagrama de objetos es un modelo abstracto de la representación en memoria, no gráfica la memoria o las direcciones en memoria, sino el estado interno de los objetos y las referencias.

Unidad 3 - Asociación y dependencia entre Clases

Los procesos de abstracción y clasificación permiten identificar, representar y agrupar a entidades que son semejantes de acuerdo a algún criterio. El criterio adoptado en una clasificación permite decidir si una entidad pertenece a una clase o no.
Las entidades de una clase son semejantes de acuerdo al criterio elegido y además se **relacionan** de la misma manera con las entidades de otras clases.
La **asociación** es una relación entre clases que se produce cuando el modelo de un objeto del problema **contiene** o puede contener al modelo de otro objeto del problema.
La relación de **dependencia** se produce cuando el modelo de un objeto del problema **usa** al modelo de otro objeto.
En la programación orientada a objetos el punto de partida para la construcción de un sistema es un proceso de abstracción y clasificación. Los objetos de una clase se caracterizan por tener

los mismos atributos y comportamiento, pero además comparten entre sí el mismo modo de relacionarse con objetos de otras clases.
Un objeto asociado a otro objeto, **tiene un** atributo de la misma u otra clase. La relación entre los objetos provoca una relación entre las clases, que se dicen **asociadas.**
Un objeto depende de un objeto de otra clase, si **usa un** objeto de esta otra clase. La relación entre los objetos provoca una relación de **dependencia** entre las clases.
Cuando una clase está asociada a otra clase, los cambios en la segunda pueden tener un impacto en la primera. El mismo impacto se produce si se modifica una clase de la cual depende otra. Uno de los objetos de la programación orientada a objetos es reducir el impacto de estos cambios.

Dependencia entre clases
Cuando una clase declara una variable local o un parámetro de otra clase, decimos que existe una **dependencia** entre la primera y la segunda y surge de una relación del tipo **usaUn** entre los objetos.

Asociación entre clases
Cuando una clase **tiene un atributo** de otra clase, ambas clases están **asociadas** y la relación es de tipo **tieneUn.** En general, entre dos clases asociadas también hay una relación de dependencia, algunos de los servicios provistos por una clase recibirán parámetros o retornarán un resultado de la clase asociada. Así, una relación de tipo **tieneUn** con frecuencia provoca una relación de tipo **usaUn.**

Igualdad y equivalencia entre objetos de clases asociadas
La representación por referencia afecta al diseño y la implementación de algunos servicios, en particular a los métodos *equals*, *copy* y *clone*. Cuando una clase está asociada a otra, la igualdad, copia y clonación se puede hacer en forma **superficial** o en **profundidad.**

Para decidir si dos objetos de una clase son iguales en forma superficial, al verificar que el estado interno de los objetos sean equivalentes se evalúa si están ligados a una misma instancia de la clase asociada. De manera análoga, la copia superficial modifica el estado interno del objeto que recibe el mensaje, con los valores almacenados en el objeto que pasa como parámetro. La clonación superficial retorna como resultado un objeto que tiene exactamente el mismo estado interno que el objeto que recibió el mensaje, incluyendo las referencias a objetos de las clases asociadas.

Cuando la igualdad es en profundidad la exigencia es menor, se requiere que los estados internos de los objetos asociados sean equivalentes, aunque las referencias sean distintas. En el caso de la copia en profundidad, el estado interno del objeto que recibe el mensaje se modifica con los valores de los atributos del objeto que se recibe como parámetro, excepto las referencias, que no se modifican, pero sí se modifica el estado interno de los objetos asociados. La clonación en profundidad retorna un nuevo objeto con el mismo estado interno que el objeto que recibe el mensaje, excepto las referencias que estarán ligadas a clones de los objetos asociados.

Los siguientes métodos implementan entonces **copia, clonación e igualdad superficial;**

```
public void copy( Alien a){
//Requiere a ligada
    nave= a.obtenerNave();
    vidas= a.obtenerVidas();
    antenas=a.obtenerAntenas();
    manos= a.obtenerManos();}
```

El comando *copy* copia el estado interno del objeto de clase *Alien* que se recibe como parámetro, en el estado interno del objeto de clase *Alien* que recibe el mensaje.

```
public Alien clone (){
    NaveEspacial n =nave;
    int  m= manos;
    int a=antenas;
    int v=vidas;
    Alien c= new Alien (n,a,m);
    c.establecerVidas(v):
    return c;}
```

La consulta *clone* retorna un nuevo objeto de clase *Alien* que referencia a la misma nave que el objeto que recibió el mensaje.

```
public boolean equals (Alien a){
//Requiere a ligada
    return nave==a.obtenerNave() && manos== a.obtenerManos() && antenas== a.obtenerAntenas() && vidas== a.obtenerVidas();}
```

La consulta *equals* retorna *true* si los valores de los atributos de los objetos que se comparan son iguales, esto implica que mantienen referencias a una misma nave o bien no están ligadas.

El código de **clone en profundidad** es:

```
public Alien clone(){
//Requiere que nave está ligada
    NaveEspacial n =nave.clone();
    int m= manos;
    int a=antenas;
    int v=vida;
  Alien c= new Alien (n,a,m);
    c.establecerVidas(v);
    return c;}
```

En esta implementación, se crea un clone del alien que recibe el mensaje, vinculado a un clone de la nave asociada al alien que recibe el mensaje. Es importante notar que si el atributo *nave* no estuviera ligado se produciría un error de ejecución. Sin embargo, si cada clase cumple con las responsabilidades establecidas en el diseño, el alien solo recibe el mensaje *clone* si está ligado a una nave.

La implementación del comando **copy en profundidad** es:

public void copy (Alien a){
//Requiere a ligada y la nave de a ligada
nave.copy(a.obtenerNave());
vidas= a.obtenerVidas();
antenas 0 a.obtenerAntenas();
manos 0 a.obtenerManos();}
La copia en profundidad modifica los valores de los atributos *vidas, antenas y manos*, y el estado interno de la nave asociada al alien que recibe el mensaje, pero no se modifica el valor del atributo instancia *nave* de la clase *Alien.*

La implementación de **equals en profundidad** es:
public boolean equals (Alien a){
//Requiere que nave está ligada
return Nave.equals(a.obtenerNave()) && manos==a.obtenerManos() && antenas== a.obtenerAntenas() && vidas == a.obtenerVidas();}

La consulta *equals* definida en la clase *Alien* envía el mensaje *equals* al objeto ligado al atributo de instancia *Nave*. Nuevamente, es la clase del objeto que recibe el mensaje la que determina cuál de los métodos debe ejecutarse en cada caso.

Clientes y proveedores de servicios
En el conjunto de clases que modelan una aplicación, algunas clases son **clientes** de los servicios provistos por otras clases **proveedoras**. Con frecuencia una misma clase puede cumplir ambos roles, es decir, ser cliente y proveedora a la vez.
Una clase que **usa** los servicios provistos por otra clase, es cliente de la clase que provee dichos servicios. Una clase que **tiene** atributos de otra clase, es cliente de las clases a las que corresponden esos atributos; estas últimas son proveedoras de servicios.
En particular una clase tester es una clase cliente de los servicios que brindan las clases que deben ser verificadas. Al implementar una clase tester se establece una dependencia entre esta clase y las que van a ser verificadas.
La clase cliente accede a la clase proveedora a través de su **interfaz.** La programación orientada a objetos propone **minimizar** la interfaz a través de la cual se comunican una clase cliente y una clase proveedora, de modo que se minimice también el impacto de los cambios.

El contrato entre la clase proveedora y la clase cliente
Entre una clase cliente y una clase proveedora de servicios, se establece un **contrato** que determina las **responsabilidades** de cada una. Las condiciones del contrato especifican en la etapa de **diseño** del sistema e incluyen la **funcionalidad** y los **requisitos** inherentes a cada servicio. En la **implementación** es necesario interpretar las responsabilidades, los requisitos y la funcionalidad para reflejarlas en el código. Parte de la **verificación** consiste en analizar si cada clase cumple con el contrato.

Cuando una clase no cumple con su contrato pueden producirse errores de ejecución o aplicación que es necesario detectar y depurar. También puede ocurrir que el contrato no se haya especificado adecuadamente. En cualquier caso es fundamental diseñar una batería de casos de prueba que permitan detectar errores. El diseño de los casos de prueba puede hacerlo el diseñador del sistema o el responsable de la etapa de testeado.

En un sistema **robusto** cada clase establece controles que previenen errores provocados por otras clases, cuando estas no cumplen con sus responsabilidades. En este caso se utilizan los mecanismos provistos por el lenguaje para **manejar excepciones.**
El diseñador del sistema establece la **responsabilidad** de cada clase. El programador debe generar código adecuado para garantizar que cada clase cumple con sus responsabilidades. El responsable del testeado busca errores de aplicación en base al contrato.
Si el diseñador elige una de las alternativas y cambia de decisión una vez que las clases están implementadas, el cambio va a requerir modificar tanto la clase proveedora, como todas las clases que la usan. Una modificación de diseño que cambia las responsabilidades, afecta a la colección de clases asociadas. Si sólo se modifica una de las clases, por ejemplo la clase cliente, va a producirse un **error de aplicación**, que pasa desapercibido para el compilador.

Unidad 4 - Encapsulamiento y Abstracción

Un sistema es una colección de componentes que interactúan entre sí con un objetivo común. En la construcción de sistemas complejos el **encapsulamiento** es un mecanismo fundamental porque permite utilizar cada componente como una "caja negra", esto es, sabiendo **qué** hace sin saber **cómo** se hace. Se reducen así las dependencias entre las diferentes componentes, cada una de las cuales es más fácil de entender, probar y modificar. El concepto de encapsulamiento está ligado a toda actividad vinculada a la producción.

Cada componente es un **módulo** que puede considerarse individualmente o como parte del conjunto. Cuando un módulo se analiza como parte de un todo, es posible concentrarse en qué función realiza, sin considerar cómo fue construido. Cuando un módulo se analiza individualmente es posible hacer abstracción de los detalles del contexto en el que va a ser utilizado, para enfocarse en cómo lograr su propósito.

En el desarrollo de sistemas de software el concepto de encapsulamiento está fuertemente ligado al de *abstracción*. Analistas y diseñadores elaboran un modelo a partir de un proceso de abstracción que especifica una colección de módulos relacionados entre sí. Los desarrolladores conservan en el código la estructura modular especificada en el diseño. Cada módulo puede ser verificado independientemente de los demás, antes de integrarse al sistema. Durante surgen cambios en la especificación o en el diseño, el encapsulamiento permite reducir el impacto.
La construcción de cada módulo de software puede realizarse sin conocer el sistema al cual va a integrarse. El sistema completo se construye a partir de una colección de módulos, sin considerar cómo se implementó cada uno en particular.

Encapsulamiento en la Programación Orientada a Objetos

En la programación orientada a objetos el concepto de encapsulamiento está ligado tanto a las clases como a los objetos. Cada objeto de software interactúa con otros objetos del entorno a través de su **interfaz** y **oculta** su estado interno y los detalles que describen cómo actúa cuando recibe un mensaje. Así, si la estructura del estado interno se modifica, el cambio no afecta a los otros miembros del entorno. Sólo las modificaciones en la interfaz pueden afectar a los demás objetos.
Para que un objeto de software oculte su estado interno, la clase que define sus atributos y comportamiento debe **esconder** la implementación. Cada clase es una caja negra, es decir, conoce **qué** hacen las demás clases del sistema, pero no **cómo** lo hacen.

El encapsulamiento es entonces el mecanismo que permite **agrupar** en una clase los atributos y el comportamiento que caracterizan a sus instancias y **esconder** la representación de los datos y los algoritmos que modelan al comportamiento, de modo que sólo sean visibles dentro de la clase.
La **estructura estática** de un sistema orientado a objetos queda establecida por una colección de clases relacionadas entre sí. El **modelo dinámico** es un entorno poblado de objetos comunicándose a través de mensajes. El comportamiento de cada objeto en respuesta a un mensaje, depende de la clase a la que pertenece.
El encapsulamiento brinda cierto nivel de independencia, cada clase puede ser construida y verificada antes de integrarse al sistema completo. Una clase puede modificar la representación de los datos o los algoritmos, sin afectar a sus clases clientes, en tanto mantenga sus responsabilidades y los servicios conserven sus compromisos y funcionalidades.

Estructura de datos
Una **estructura de datos** es una colección de datos organizados de alguna manera. Los atributos de una clase definen una estructura de datos que permite representar el estado interno de los objetos de esa clase en ejecución. La programación orientada a objetos propone encapsular la representación de los datos, de modo que no sean visibles desde el exterior de la clase.
En Java los **modificadores de acceso** permiten establecer la visibilidad de los miembros de una clase. Los atributos se declaran privados y quedan así escondidos dentro de la clase.

Representaciones alternativas para estructuras lineales y homogéneas
En la resolución de problemas de mediana y gran escala el diseño de las estructuras de datos es un tema relevante. Cuando una clase modela una secuencia homogénea de elementos, por lo general existen diferentes estructuras alternativas para representar tanto a los elementos como a la colección misma. Si cada servicio de la clase mantiene la signatura, además de los requisitos y funcionalidad, los cambios en la representación de los datos no modifican la interfaz, aunque sí va a ser necesario modificar el código.

Vamos a utilizar **arreglos** para modelar estructuras **lineales**, **homogéneas** y con **acceso directo**. Una estructura es **homogénea** si todos los elementos son del mismo tipo. En una estructura **lineal** cada elemento tiene un predecesor, excepto el primero, y un sucesor, excepto el último. Una estructura tiene **acceso directo** si cada una de sus componentes puede accederse a través de su posición, sin necesidad de acceder a las que la preceden o suceden.

Encapsulamiento y clases relacionadas
La modificación de la representación de los datos en una clase implica reescribir el código de algunos o incluso todos los servicios. Si los atributos están escondidos, el cambio en la representación es transparente para las clases asociadas, en tanto no cambien las responsabilidades de la clase ni la funcionalidad o los requisitos de los servicios.

Abstracción y programación orientada a objetos
El análisis y diseño orientado a objetos demanda un proceso de abstracción a partir del cual se identifican los objetos del problema y se agrupan en clases. Una clase es una abstracción, un patrón que caracteriza los atributos y el comportamiento de un conjunto de objetos.

En la implementación, una clase que establece atributos y servicios define un **tipo de dato** a partir del cual es posible crear instancias. El conjunto de valores queda determinado por el tipo de los atributos y el conjunto de operaciones corresponde a los servicios provistos por la clase. Si la representación de los datos está escondida, la clase define un **tipo de dato abstracto (TDA).**

La definición de tipos de datos abstractos es un recurso importante porque favorece la reusabilidad y permite reducir el impacto de los cambios. La modificación de una clase que define un tipo de dato abstracto, puede realizarse sin afectar a las clases que crean instancias del tipo.
En Java la definición de clases y los modificadores de accesos, permiten que el programador defina nuevos tipos de datos abstractos a partir de los cuales se crean instancias.

TDA y estructuras parcialmente ocupadas
La abstracción permite proponer soluciones similares para problemas en apariencia muy diferentes. Esas similitudes van a permitir reusar diseño y código, favoreciendo la productividad.
Utilizaremos el término **colección** para referirnos a una estructura con capacidad para mantener max elementos de un **tipo base**. En cada momento determinado sólo n de los max elementos referencian a un objeto del tipo base. Los n elementos ligados ocupan las primeras n posiciones de la estructura. De modo que las max-n componentes nulas, están comprimidas en las últimas posiciones del arreglo. Si la posición de cada elemento en una colección no tiene relevancia, puede cambiar de posición siempre y cuando todas las n referencias ligadas se mantengan en las primeras n posiciones.
Utilizaremos el término **tabla** para referirnos a una estructura con n elementos de un tipo base, cada uno de los cuales ocupa una posición que es significativa en la estructura. Es decir las max-n componentes nulas pueden estar intercaladas con las componentes ligadas.

Cuando una colección de elementos está ordenada de acuerdo a un atributo y es necesario decidir si un elemento en particular pertenece a la colección, es posible aplicar una estrategia conocida como **búsqueda binaria.** La estrategia consiste en partir la estructura en mitades, considerando que el **elemento buscado** puede ser:

- ★ Igual al que está en el medio
- ★ Menor que el que está en el medio
- ★ Mayor que el que está en el medio

De modo que podemos proponer un algoritmo recursivo:

```
Algoritmo Búsqueda Binaria
Dato de entrada: elem
    si el elemento que está en el medio es el buscado
    EXISTE
    si hay un sólo elemento y no es el buscado
        NO EXISTE
  sino
        si el elemento que está en el medio es menor al buscado
            Descartar la primera mitad
            Buscar en la segunda mitad
        sino
            Descartar la segunda mitad
```

Buscar en la primera mitad

Unidad 5 - Herencia y Polimorfismo

En un **modelo basado** en clases las entidades se agrupan a partir de sus semejanzas y diferencias, establecidas en función de algún criterio. Una vez establecido el **criterio de clasificación**, es posible decidir si una entidad **pertenece** a una clase.
La relación de pertenencia vincula a una entidad con una clase. La relación de inclusión vincula a una clase con otra. Una entidad que pertenece a una clase, pertenece también a cualquier clase que la incluya.
La clasificación en niveles es una habilidad natural para el ser humano y permite modelar la relación de **herencia** entre clases. Una clase específica hereda las propiedades de las clases más generales, en las cuales está incluida
En la programación orientada a objetos la **herencia** es el mecanismo que permite crear clasificaciones que modelan una relación de **generalización-especialización** entre clases. Las clases especializadas heredan atributos y comportamiento de las clases generales y agregan atributos y comportamiento específico. Una clase especializada puede también redefinir el comportamiento establecido por su clase más general.
El concepto de herencia está fuertemente ligado al de polimorfismo. El **polimorfismo** permite que las diferentes entidades de una misma clase puedan exhibir distintas formas de un mismo comportamiento. En la programación orientada a objetos el **polimorfismo** es el mecanismo que permite que un mismo nombre pueda quedar ligado a objetos de diferentes clases, de modo que objetos de distintas clases puedan recibir un mismo mensaje y cada uno actúe de acuerdo al comportamiento establecido por su clase.
En el desarrollo de un sistema de software la herencia es un recurso importante porque favorece la **reusabilidad** y la **extensibilidad**. El polimorfismo también favorece la productividad porque permite que un mismo nombre pueda asociarse a abstracciones diferentes, dependiendo del contexto.

Herencia

Una clase es un patrón que define los atributos y comportamiento de un conjunto de entidades. Dada una clase es posible definir otras más específicas que heredan los atributos y comportamiento de la clase general y agregan atributos y comportamiento especializado.

La **herencia** es un mecanismo que permite organizar la colección de clases de un sistema, estableciendo una relación de **generalización-especialización.**

Cuando el diseño propone dos o más clases que comparten algunos atributos y servicios y difieren en otros, es posible definir una **clase base** y una o más **clases derivadas**. La clase base especifica los atributos y servicios compartidos. Las clases derivadas o subclases especializan a la clase base, heredan atributos y comportamiento de las clases generales y agregan atributos y comportamiento específico. Una clase derivada puede también redefinir el comportamiento establecido por su clase más general.

Un objeto pertenece a una clase si puede ser caracterizado por sus atributos y comportamiento.
Una clase es **derivada** de una clase **base**, si todas sus instancias pertenecen también a la clase base.

La herencia es un recurso poderoso porque favorece la extensibilidad. Con frecuencia los cambios en la especificación del problema se resuelven incorporando nuevas clases especializadas, sin necesidad de modificar las que ya han sido implementadas, verificadas e integradas al sistema. La herencia facilita la reusabilidad porque no sólo se reutilizan clases, sino colecciones de clases relacionadas a través de herencia.

Diseño orientado a objetos

El diseño orientado a objetos consiste en definir una colección de clases relacionadas entre sí. Las clases pueden relacionarse a través de:

- **Asociación:** permite modelar la relación **tieneUn**, esto es, un atributo de instancia de una clase corresponde a otra clase.
- **Dependencia:** permite modelar la relación **usaUn**, es decir, los métodos de una clase reciben parámetros o declaran variables locales de otra clase.
- **Herencia:** permite modelar la relación de generalización-especialización de tipo **esUn**, las instancias de una clase derivada son también instancias de las clases de las cuales hereda.

La clasificación agrupa objetos de un conjunto de acuerdo a un criterio, definiendo una colección de clases. La herencia aumenta el nivel de abstracción porque las clases son a su vez clasificadas a partir de un proceso de **generalización** o **especialización**. Si hablamos de abstracción cuando agrupamos objetos en clases, podemos llamar **superabstracción** al proceso de clasificar clases.

El proceso de clasificación puede hacerse partiendo de una clase muy general y descomponiéndola en otras más específicas identificando las diferencias entre los objetos. Si el proceso continúa hasta alcanzar subclases homogéneas, hablamos de **especialización**. Alternativamente es posible partir del conjunto de todos los objetos y agruparlos en clases según sus atributos y comportamiento. Estas clases serán a su vez agrupadas en otras de mayor nivel hasta alcanzar la clase más general. Hablamos entonces de **generalización**.

Herencia simple y herencia múltiple

Cuando la **herencia es simple** la clasificación es jerárquica y queda representada por un **árbol**. En este caso el proceso de clasificación se realiza de manera tal que cada subclase corresponde a una única clase base. Cada clase puede derivar entonces en una o varias subclases o clases derivadas, pero sólo puede llegar a tener una única clase padre. La raíz del árbol es la clase más general, las hojas son las clases más específicas. El término superclase en ocasiones se usa para referirse a la raíz y otros autores lo utilizan como sinónimo de clase base.

Las clases **descendientes** de una clase son las que heredan de ella directa o indirectamente, incluyéndola a ella misma. Los **descendientes propios** de una clase son todos sus descendientes, excepto ella misma.

El conjunto de clases **ancestro** de una clase, incluye a dicha clase y a todas la que ocupan los niveles superiores en la misma rama del árbol que grafica la estructura de herencia. Los **ancestros propios** de una clase son todos sus ancestros, excepto ella misma.

Las **instancias** de una clase son los objetos que son instancia de alguna clase descendiente de dicha clase. Las **instancias propias** de una clase son los objetos de dicha clase.

La **herencia múltiple** permite que una clase derivada pueda heredar de dos o más clases más generales. Es una alternativa poderosa pero más compleja. Nuevamente las clases de los niveles superiores son más generales que las clases de los niveles inferiores.

Herencia y constructores

Una clase derivada hereda de la clase base todos sus atributos y métodos, pero no los constructores. Cada clase derivada tendrá sus propios constructores, que pueden invocar a los constructores de las clases más generales.

Si se invoca al constructor de una clase base mediante *super*, siempre tiene que ser en la primera línea de la definición del constructor de la clase derivada. En ejecución, el estado interno de un objeto de una clase derivada mantiene los atributos propios de la clase y todos los atributos heredados de las clases ancestro.

Herencia y extensibilidad

La extensibilidad es una cualidad fundamental para lograr productividad. La herencia permite que, con frecuencia, los cambios en los requerimientos puedan resolverse definiendo nuevas clases, sin modificar las clases que ya han sido desarrolladas y verificadas.

Polimorfismo

Polimorfismo significa muchas formas y en Ciencias de la Computación en particular se refiere a la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál corresponde usar. El concepto de polimorfismo es central en la programación orientada a objetos y se complementa con el mecanismo de herencia.

El polimorfismo es el mecanismo que permite que objetos de distintas clases puedan recibir un mismo mensaje y cada uno actúe de acuerdo al comportamiento establecido por su clase.

En el contexto de la programación orientada a objetos el polimorfismo está relacionado con variables, asignaciones y métodos.

- Una **variable polimórfica** puede quedar asociada a objetos de diferentes clases.
- Una **asignación polimórfica** liga un objeto de una clase a una variable declarada de otra clase.
- En un **método polimórfico** al menos uno de los parámetros formales es una variable polimórfica.

Dado que una variable puede estar asociada a objetos de diferentes tipos, es posible distinguir entre:

- El **tipo estático** de una variable, es el tipo que aparece en la declaración.
- El **tipo dinámico** de una variable es la clase a la que pertenece el objeto referenciado.

El tipo estático lo determina el compilador, el tipo dinámico se establece en ejecución.

Redefinición de métodos

Un lenguaje que soporta el concepto de polimorfismo permite establecer que todos los objetos de una clase base van a brindar cierta funcionalidad, aunque la forma de hacerlo va a depender de su clase específica.

En Java un mismo nombre puede utilizarse para definir un método en la clase base y otro en la clase derivada. Si en la clase derivada se define un método con el mismo nombre, número y tipo de parámetros que un método definido en la clase base, el método de la clase base queda **derogado**. La definición del método en la clase derivada **redefine** el método de la clase base.

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida. En Java se utiliza entonces el **modificador** *final*, que tiene significados levemente distintos según se aplique a una variable, a un método o a una clase.

- ❖ Para una **clase**, *final* significa que la clase no puede extenderse. Es, por tanto, una hoja en el árbol que modela la jerarquía de clases.
- ❖ Para un **método**, el modificador *final* establece que no puede redefinirse en una clase derivada
- ❖ Para un atributo, *final* establece que no puede ser redefinido en una clase derivada, pero además su valor no puede ser modificado.

Cuando en una clase se define un método con el mismo nombre que otro de la misma clase o de alguna clase ancestro, pero con diferente número o tipo de parámetros, el método queda **sobrecargado**.

Ligadura dinámica de código

La ligadura dinámica de código es la vinculación en ejecución de un mensaje con un método. Esto es, cuando un método definido en una clase, queda redefinido en una clase derivada, el tipo dinámico de la variable determina qué método va a ejecutarse en respuesta a un mensaje. El compilador no puede establecer la ligadura, es en ejecución que se resuelve qué mensaje corresponde ejecutar. Polimorfismo, redefinición de métodos y ligadura dinámica de código, son conceptos fuertemente relacionados. La posibilidad de que una variable pueda referenciar a objetos de diferentes clases y que un método pueda ser redefinido en las clases derivadas, brinda flexibilidad al lenguaje, siempre que además exista ligadura dinámica de código.

Chequeo de tipos

El polimorfismo es un mecanismo que favorece la **reusabilidad** pero debe restringirse para brindar **confiabilidad**. Los chequeos de tipos en compilación garantizan que no van a producirse errores de tipo en ejecución.

El chequeo de tipos establece restricciones sobre:

- ★ Las asignaciones polimórficas
- ★ Los mensajes que un objeto puede recibir

En una asignación polimórfica, la clase del objeto que aparece a la derecha del operador de asignación, debe ser de la misma clase o de una clase descendiente de la clase de la variable que aparece a la izquierda del operador. Así, el tipo estático de una variable determina el conjunto de tipos dinámicos.

El chequeo de tipo pretende justamente establecer controles en compilación que eviten errores de ejecución.

Con respecto a restricciones sobre los mensajes, un objeto sólo puede recibir mensajes para los cuales existe un método definido en la clase que corresponde a la declaración de la variable, o en sus clases ancestro.

El tipo estático de la variable determina los mensajes que un objeto puede recibir, pero el tipo dinámico determina la implementación específica del comportamiento que se ejecuta en respuesta a los mensajes. Es decir, el compilador chequea la validez de un mensaje considerando el tipo estático de la variable. En ejecución, el mensaje se liga con el método, considerando el tipo dinámico.

Casting

Casting es un mecanismo provisto por Java para relajar el control del compilador. El programador se hace responsable de garantizar que una asignación va a ser válida o un mensaje va a poder ligarse. Si se produce un mal casting se producirá un error en ejecución, esto es, una terminación anormal.

Clases abstractas

En el diseño de una aplicación es posible definir una clase que factoriza propiedades de otras clases más específicas, sin que existan en el problema objetos concretos vinculados a esta clase más general. En este caso la clase se dice **abstracta** porque fue creada para lograr un modelo más adecuado. En ejecución no va a haber objetos de software de una clase abstracta.

Una clase abstracta puede incluir uno, varios, todos o ningún **método abstracto**. Un método abstracto es aquel que no puede implementarse de manera general para todas las instancias de la clase. Una clase que incluye un método abstracto define sólo su signatura, sin bloque ejecutable. Esto es, todos los objetos de la clase van a ofrecer una misma funcionalidad, pero la implementación concreta no puede generalizarse.

Si una clase hereda de una clase abstracta y no implementa todos los métodos abstractos, también debe ser definida como abstracta. El constructor de una clase abstracta sólo va a ser invocado desde los constructores de las clases derivadas. Una clase concreta debe implementar todos los métodos abstractos de sus clases ancestro.

El modificador **abstract** permite declarar una clase abstracta

Unidad 6 - Genericidad

La generalización consiste en abstraer lo que es común a un conjunto de entidades para definir un concepto que las incluya a todas. La generalización es una estrategia fundamental en la resolución de problemas de diferentes disciplinas, en particular en Ciencias Exactas, Ciencias Naturales y en Ciencias de la Computación. El concepto de **genericidad**, muy relacionado con la estrategia de generalización, se refiere a todo aquello que es común entre los miembros de una especie y permite por lo tanto establecer un patrón común.

En programación, un algoritmo es genérico si su código no depende del tipo de los datos. En numerosas aplicaciones es posible representar los datos con estructuras que pueden ser procesadas por algoritmos genéricos, esto es, el código no depende del tipo de las componentes de la estructura.

La genericidad es un recurso poderoso porque favorece la extensibilidad y la reusabilidad. Los datos de aplicaciones muy diferentes pueden modelarse con estructuras que serán creadas, modificadas y procesadas sin considerar el tipo de las componentes. La extensibilidad reduce el impacto de los cambios. Las modificaciones con frecuencia pueden resolverse definiendo nuevas clases específicas, sin necesidad de cambiar las que ya han sido desarrolladas y verificadas. La reusabilidad evita escribir el mismo código repetidamente acelerando el proceso de desarrollo.

Genericidad y Programación Orientada a Objetos

La programación orientada a objetos tiene como principal objetivo favorecer la confiabilidad, reusabilidad y extensibilidad del software. Adoptar el enfoque propuesto por la programación orientada a objetos implica:

- ★ En la etapa de diseño reducir la complejidad en base a la descomposición del problema en piezas más simples, a partir de un conjunto de clases relacionadas entre sí.
- ★ En la etapa de implementación utilizar un lenguaje que permita retener las relaciones entre las clases y encapsular su representación interna.

La abstracción de datos y el encapsulamiento permiten que una o más clases usen los servicios provistos por otra clase considerando sólo su comportamiento, sin tener en cuenta cómo lo implementa. La herencia permite aumentar el nivel de abstracción mediante un proceso de clasificación en niveles. La herencia puede utilizarse para modelar genericidad.

Alternativamente la genericidad puede modelarse usando polimorfismo paramétrico. El polimorfismo paramétrico en Java es limitado y tiene varias restricciones.

Clases genéricas
Una **clase genérica** es aquella que encapsula a una estructura cuyo comportamiento es independiente del tipo de sus elementos.

Generalización y reuso
Al analizar el diagrama de clases de distintos sistemas es posible observar que un conjunto de servicios brindan una funcionalidad equivalente, de modo que se favorece el reuso definiendo una clase genérica que los incluya.

Servicios generales de una clase
Una clase genérica brinda un conjunto de servicios generales cuya implementación es independiente del tipo de las componentes de la estructura. La clase genérica puede ser usada para modelar diferentes aplicaciones.

Servicios específicos de una clase
En muchas aplicaciones es posible reusar una clase genérica, pero es necesario extenderla con otra que brinde el comportamiento específico.

Unidad 7- Interfaces Gráficas de Usuario
La programación orientada a objetos favorece la construcción y mantenimiento de sistemas de software complejos desarrollados para ambientes dinámicos. El encapsulamiento, la herencia y el polimorfismo permiten producir componentes reusables y fáciles de extender, como las que se requieren cuando se desarrolla una **interfaz gráfica de usuario**.
Una **interfaz** es un medio que permite que dos entidades se comuniquen. En una **interfaz de usuario** una de las entidades es una **persona** y la otra es un **sistema** que el usuario intenta controlar.
Las **interfaces gráficas de usuario (GUI)** explotan la capacidad de las computadoras para reproducir imágenes y gráficos en pantalla y brindan un ambiente amigable, simple e intuitivo. Como su nombre lo indica, los tres elementos que definen a un GUI son:

- ★ **Interfaz:** medio de comunicación entre entidades.
- ★ **Gráfica:** incluye ventanas, menúes, botones, texto, imágenes, etc.
- ★ **Usuario:** persona que usa la interfaz para controlar un sistema.

Las GUI reemplazaron a las interfaces de líneas de comandos en la cuales el usuario interactuaba con el sistema a través de una consola. En una GUI la interacción se realiza a través de:

- ❖ Iconos
- ❖ Ventanas
- ❖ Menúes desplegables
- ❖ Hipertexto
- ❖ Manipulación basada en *arrastrar y soltar*

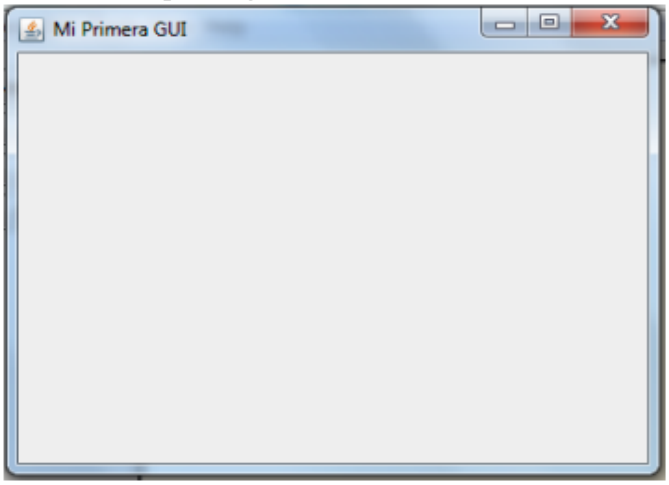
Un **ícono** es un pictograma que representa a una entidad como un archivo, una carpeta, una acción o una aplicación.
Una **ventana** es una porción de la pantalla que sirve como una pantalla más pequeña. Un **menú** es una lista de opciones alternativas ofrecidas al usuario. Un menú desplegable es aquel que "cuelga" de una opción de otro menú.

Un **hipertexto** es un texto organizado mediante una red. Si el texto se organiza en red y además incluye imágenes, sonido, animación, se llama **hipermedio**.
La **manipulación basada en arrastrar y soltar** presupone el uso del ratón como dispositivo de entrada asociado a un cursor.
Una GUI se construye a partir de una **colección de componentes** con una **representación gráfica**. Por ejemplo, el botón para cerrar una ventana, la barra de desplazamiento de una ventana y la ventana misma.
Algunas componentes tienen la capacidad para **percibir eventos** generados por las acciones del usuario. Un usuario realiza una **acción** que genera un **evento** que es detectado por una **componente** lo cual provoca una **reacción**. La creación de componentes reactivas requiere que el lenguaje brinde algún mecanismo para el **manejo de eventos**. El **flujo de ejecución** va a quedar determinado por los eventos que generan los usuarios sobre las componentes de la interfaz.
La construcción de GUI en Java requiere integrar los conceptos centrales de la POO: **herencia, polimorfismo, ligadura dinámica y encapsulamiento.** El uso de paquetes gráficos brinda la oportunidad de reusar y extender clases recursos provistos por el lenguaje.

Objetos gráficos y Clases gráficas
Las componentes son las partes individuales a partir de las cuales se conforma una interfaz gráfica. Cada componente de una GUI está asociada a un **objeto gráfico**. Un objeto gráfico es una instancia de una **clase gráfica**. Algunos de los atributos de un objeto gráfico son atributos gráficos y parte del comportamiento ofrece **servicios gráficos**.
Algunos componentes son **contenedores** de otros componentes. Un contenedor tiene atributos especiales como por ejemplo el **diagramado** de acuerdo al cual se organizan las componentes contenidas.
Una **ventana** es un contenedor de alto nivel. Un **frame** es un tipo especial de ventana sobre el que puede ejecutarse una aplicación. Un frame tiene atributos gráficos como tamaño, barra de título, marco, panel de contenido y tres botones. Cada una de estas componentes tiene también sus atributos gráficos.
La ejecución del siguiente segmento de código:

```
import javax.swing.*;
class PrimerEjemplo {
   public static void main(String args[ ]) {
            JFrame f = new JFrame("Mi Primera GUI");
      f.setSize(400, 300) ;
            f.setVisible(true) ; }}
```

Crea un frame como el que sigue:



La creación de un objeto de la clase *JFrame* requiere importar el paquete Swing. La instrucción:

JFrame f = new JFrame("Mi Primera GUI");

crea un objeto de la clase *JFrame* usando el constructor que recibe una cadena de caracteres con la que se inicializa el título de la barra de título. La clase *JFrame* brinda servicios para modificar los atributos gráficos. La variable *f* mantiene una referencia a un objeto de clase *JFrame* y puede recibir cualquiera de los mensajes provistos en esa clase o en sus clases ancestro. Las instrucciones:

f.setSize(400, 300) ;
f.setVisible(true) ;

envían mensajes al frame para establecer su tamaño y hacerlo visible. Un efecto equivalente para el usuario, se consigue definiendo una clase que extiende a *JFrame* y creando un objeto de esta clase derivada.

```
import java.awt.*;
import javax.swing.*;
class MiVentana extends JFrame{
      public MiVentana() {
      super("Mi ventana");
            setSize(400, 300);
            getContentPane().setBackground(Color.BLUE);
            setDefaultCloseOperation(EXIT_ON_CLOSE);}}
```

La instrucción *super("Mi ventana")* invoca al constructor provisto por *JFrame* con la cadena que aparecerá en la barra de título como parámetro.

La instrucción *setSize(400, 300)* establece el ancho y la altura del frame en pixels.

El mensaje *getContentPane()* obtiene el panel de contenido del frame, el objeto que retorna es de clase *Container*. Este objeto recibe a continuación el mensaje *setBackground(Color.BLUE)* cuyo efecto es establecer el color del fondo.

El mensaje *setDefaultCloseOperation* se va a ligar a un método provisto por *JFrame*. El parámetro indica qué hacer cuando la ventana se cierra, en este caso terminar la aplicación.

El constructor envía tres mensajes al mismo objeto que se está creando, la instancia de *MiVentana*, que es también instancia de *JFrame*.

El método *main* crea ahora una instancia de *MiVentana*:

```
import javax.swing.*;
class SegundoEjemplo {
   public static void main(String args[ ]) {
            MiVentana f= new MiVentana();
            f.setVisible(true); }
}
```

Como antes, la creación del objeto ejecuta el constructor y activa el frame. El objeto ligado a *f* puede recibir cualquiera de los mensajes provistos por la clase *JFrame*. La barra de título y los botones son componentes **reactivas**, reaccionan con un click del mouse.

Construcción de una GUI

La construcción de una GUI requiere:

- ➢ **Diseñar** la interfaz de acuerdo a las especificaciones
- ➢ **Implementar** la interfaz usando las facilidades provistas por el lenguaje

El **diseño** de una interfaz gráfica abarca tres aspectos:

- ★ Definir las componentes

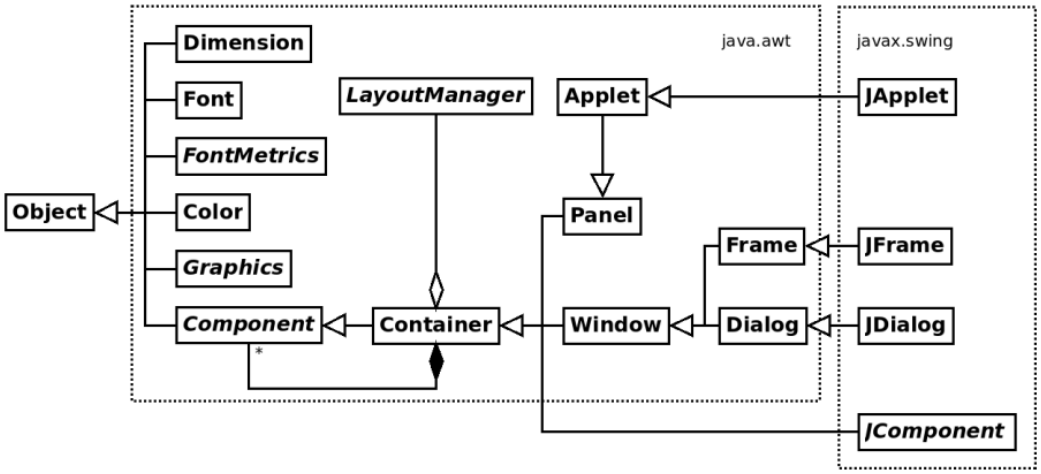
- ★ Organizar las componentes estableciendo el diagramado de las componentes contenedoras
- ★ Decidir cómo reacciona cada componente ante las acciones realizadas por el usuario

La **implementación** de una interfaz gráfica consiste en:

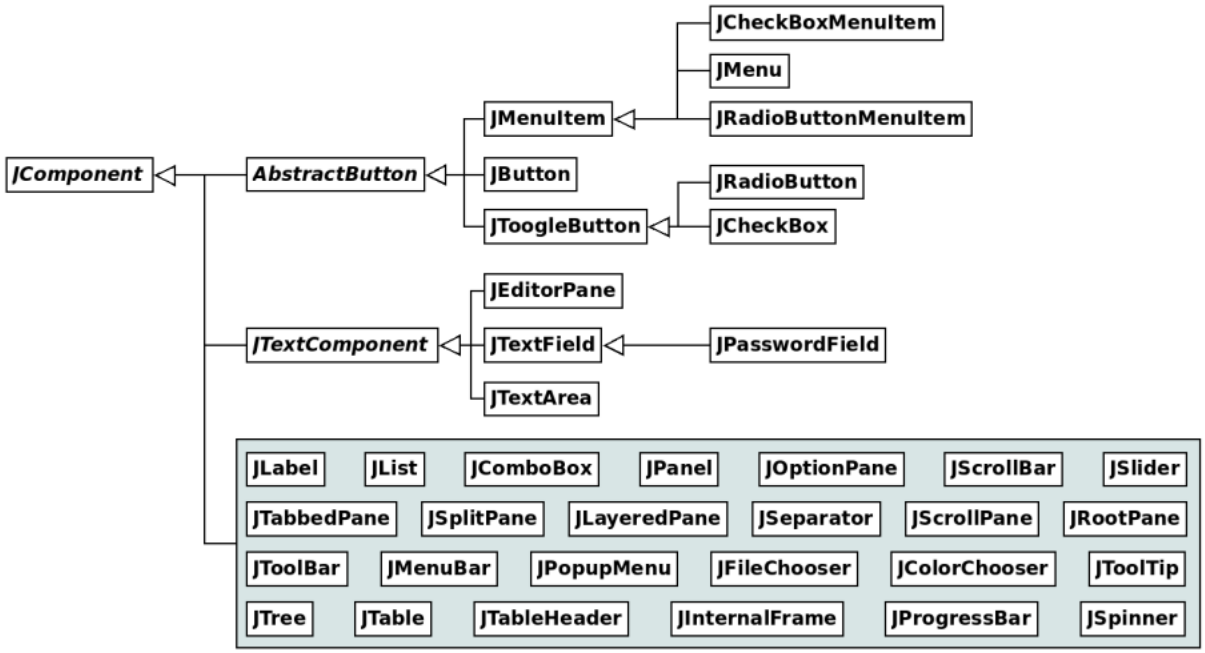
- ➢ **Crear** un objeto gráfico para cada componente de la GUI e **insertarlo** en otras componentes contenedoras.
- ➢ **Definir el comportamiento** de las componentes reactivas en respuesta a las acciones del usuario

Tipos de Componentes

Las **componentes** de las GUI que se desarrollan en los ejemplos que siguen, son instancias de alguna de las clases provistas por los paquetes gráficos AWT (Abstract Window Toolkit) o Swing o de una clase derivada de ellas. AWT y Swing son paquetes gráficos independientes de la plataforma, que permiten desarrollar interfaces gráficas. Swing no reemplaza a AWT, sino que lo usa y agrega nuevas clases, como muestra la figura.



Ambos paquetes brindan una colección de clases que pueden usarse para crear distintos **tipos de componentes**, como por ejemplo, botones, cajas de texto, menúes, etc. La jerarquía de clases que modela los diferentes tipos de componentes es:



Cada clase provista por AWT o Swing puede ser usada para:

- ➢ Crear objetos gráficos asociados a las componentes de la interfaz.
- ➢ Definir clases más específicas a partir de las cuales se crearán componentes,

Cada componente tiene una apariencia estándar pero también puede configurarse de acuerdo a las pautas de diseño que se adopten. El tamaño de cada componente se mide en pixels. Un **pixel** es la unidad de espacio más pequeña que puede mostrarse en pantalla. La **resolución** de una pantalla se mide de acuerdo a la cantidad de pixels que puede mostrar. Cuanto mayor sea la cantidad de pixels en la ventana, mayor será la resolución.

Etiquetas

Una **etiqueta** es un objeto gráfico **pasivo** que permite mostrar un texto o una imagen. En Java se puede crear una etiqueta definiendo un objeto de clase *JLabel*. Los constructores provistos por la clase permiten establecer texto, imagen y/o alineación horizontal de la imagen:

JLabel()
JLabel (Icon image)
JLabel (Icon image, int horAlig)
JLabel (String text)
JLabel (String text, Icon icon, int horAlig)

El panel de contenido es un atributo de instancia del frame y tiene a su vez atributos que pueden modificarse, como por ejemplo el color.

Cuando varias componentes se insertan en el panel de contenido, una queda superpuesta sobre la otra. Para distribuirlas de modo que todas queden visibles es necesario establecer el **diagramado** y también definir un tamaño adecuado para el frame.

El **organizador de diagramado** es un atributo de todos los objetos gráficos contenedores que determina como se distribuyen las componentes contenidas. Algunas de las clases provistas para crear organizadores son *BorderLayout*, *FlowLayout*, *GridLayout*.

- *FlowLayout:* Distribuye los componentes uno al lado del otro comenzando en la parte superior. Por omisión provee una alineación centrada, pero también puede alinear a la izquierda o derecha.
- *BorderLayout:* Divide el contenedor en cinco regiones: NORTH, SOUTH, EAST, WEST y CENTER, admite un único componente por región.
- *GridLayout*: Divide el contenedor en una grilla de n filas por m columnas, con todas las celdas de igual tamaño.

Paneles

Aunque es posible insertar las componentes gráficas directamente sobre el panel de contenido del frame, resulta más sencillo organizarlas en paneles, esto es, instancias de la clase *JPanel()*. Un **panel** es un contenedor de otras componentes gráficas. Los paneles se organizan en forma jerárquica. El principal atributo de un panel es el organizador de diagramado que establece cómo se distribuyen las componentes en su interior, tal como sucede con el panel de contenido. Así, es posible establecer un organizador de diagramado para el panel de contenido y luego insertar en él dos o más paneles. Cada uno de estos paneles puede tener un diagramado diferente. En cada uno de ellos se insertan componentes que se distribuyen de acuerdo al diagramado establecido.

Botones

Un **botón** es una componente **reactiva**, esto es, puede percibir la acción del usuario y reaccionar de acuerdo al comportamiento establecido por el programador. Un botón se crea como una instancia de la clase *JButton*.

Como las etiquetas, sus atributos principales son texto e imagen. El texto y la imagen tienen atributos alineación vertical y horizontal. Cada botón tiene también un **color**, un **borde**, una **letra mnemónica** y una **forma**, y además puede estar habilitado o no.

Algunos de los constructores provistos para *JButton* son:
JButton ()
JButton (Icon image)
JButton (String text)
JButton (String text, Icon icon)
La definición de métodos internos y los comentarios muestran la estructura del código y favorecen la legibilidad.
//Obtiene el panel de contenido y crea los objetos gráficos
//Establece los atributos del frame
//Establece los atributos de los paneles
//Crea un oyente para cada boton y lo registra
//Inserta la etiqueta y los botones en los paneles
//Inserta los paneles en el panel de contenido
El panel de contenido del frame se organiza como una grilla de dos filas y una única columna. Los objetos ligados a *panelImagen* y *panelBotones* son instancias de la clase *JPanel* y en ambos el diagramado es un objeto de clase *FlowLayout*. Las componentes se disponen en una fila de un tamaño preestablecido. Si el tamaño es mayor que el requerido, por omisión las componentes quedan centradas.
Una **fuente de evento** es un objeto que está asociado a una componente gráfica y puede percibir una acción del usuario.
Un **oyente** es un objeto que espera que ocurra un evento y **reacciona** cuando esto sucede.
Java brinda otras clases para crear botones, como por ejemplo *JRadioButton* que permite crear botones que pueden agruparse de modo tal que solo uno puede estar seleccionado. Los atributos de instancia de la clase son: el texto, el ícono y el estado.
Los constructores de un objeto de clase *JRadioButton* son:
JRadioButton(String s)
JRadioButton(String s, boolean b)
JRadioButton(Icon i)
JRadioButton(Icon i, boolean b)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean b)
JRadioButton()
El parámetro de tipo *boolean* establece si el botón está seleccionado.

Cajas de opciones
Una **caja de opciones** puede crearse como una instancia de la clase *JComboBox*, editable o no editable. Una caja de opciones no editable consta de una lista de valores drop-down y un botón. Una caja de opciones editable tiene además un campo de texto con un pequeño botón. El usuario puede elegir una opción de la lista o tipear un valor en el campo de texto. El constructor recibe como parámetro un arreglo de objetos de clase *String*.

Campos de Texto
Un **campo de texto** es una caja que permite ingresar una línea de texto por teclado. Un objeto de la clase *JTextField* permite mantener un campo de texto. Los atributos son una cadena de caracteres, la cantidad de caracteres que se visualizan en la caja y el modelo a utilizar. Cada vez que el usuario tipea una tecla se crean objetos de clase *KeyEvent* o de la clase *ActionEvent.*
Los constructores provistos por la clase son:
JTextField()

JTextField(Document doc, String text, int col)
JTextField(int col)
JTextField(String text)
JTextField(String text, int col)

Paneles de diálogo

Un **panel de diálogo** se usa para leer un valor simple y/o mostrar un mensaje. Los atributos incluyen uno o más botones. El mensaje puede ser un error o una advertencia y puede estar acompañado de una imagen o algún otro elemento. Para definir un diálogo estándar se usa la clase *JOptionPane* que brinda los siguientes servicios:

showMessageDialog: Muestra un diálogo modal con un botón, etiquetado "OK". Se puede especificar el icono y el texto del mensaje y del título. Por omisión el ícono es de información

showConfirmDialog: Muestra un diálogo modal con dos botones, etiquetados "Yes" y "No". Por omisión aparece el ícono question.

showOptionDialog: Requiere especificar el texto de los botones.

showInputDialog: Muestra un diálogo modal que obtiene una cadena del usuario. La cadena puede ser ingresada por el usuario en un campo de texto o elegida de un ComboBox no editable.

En cada caso se puede utilizar un icono personalizado, no utilizar ninguno, o utilizar uno de los cuatro iconos estándar de *JOptionPane* (question, information, warning, y error). Por omisión aparece el ícono *question*.

Todo diálogo es dependiente de un frame. Cuando un frame se destruye, también se destruyen los diálogos que dependen de él. Un diálogo es modal cuando bloquea la entrada de datos del usuario a través de todas las demás ventanas.

Los cuadros de diálogo creados con *JOptionPane* son modales. Para crear un diálogo no modal es posible usar la clase *JDialog*.

Lectura y escritura

Toda la lectura y escritura realizada por componentes de la clase Swing se realiza usando objetos de clase *String*. La entrada y salida de números requiere entonces convertir valores. La conversión de un valor de tipo *int* a *String* puede implementarse concatenando dos cadenas:

```
String s = ""+42;
```

La conversión de una cadena a un entero requiere usar el método *parseInt* de la clase estática *Integer*:

```
String s = "42";
int num = Integer.parseInt(s);
```

En forma análoga es posible convertir una cadena de caracteres al tipo double:

```
double val = Double.parseDouble("42.5");
```

La estructura de una GUI

La estructura de una GUI debería favorecer la legibilidad de modo que el código sea fácil de verificar y mantener. El código de cada una de las clases que implementan una GUI incluye:

- Instrucciones para importar paquetes gráficos
- Declaraciones de atributos de instancia
- Definición de un constructor y algunos métodos internos invocados por el constructor
- Definición de clases internas que implementan interfaces y permiten crear oyentes

Java permite definir clases oyentes anónimas e implementar el método *actionPerformed* directamente en el parámetro real del mensaje *addActionListener*. Esta alternativa compromete la legibilidad del código.
También es posible implementar las clases oyentes a través de clases externas. Esta es una alternativa adecuada para definir interfaces gráficas complejas.
El constructor o los métodos internos invocados por el constructor incluyen instrucciones para:

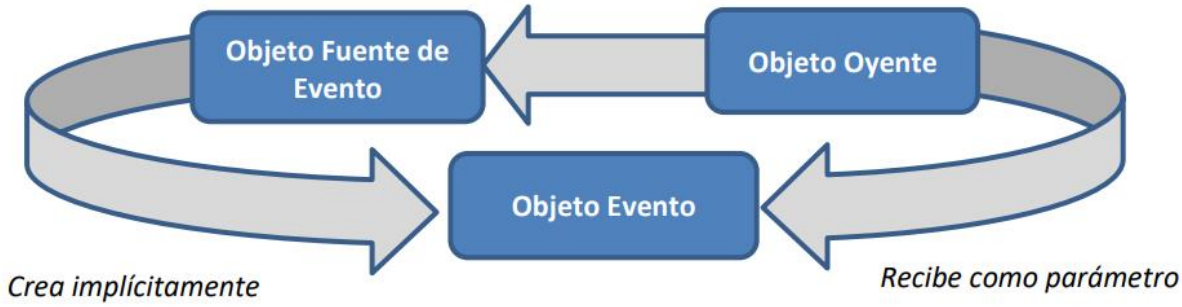
- Establecer los valores de los atributos gráficos del frame
- Crear objetos gráficos y establecer los valores de sus atributos, algunos de estos objetos serán paneles contenedores
- Crear objetos oyentes y registrarlos a los objetos asociados a componentes reactivas
- Insertar los objetos gráficos en los paneles y los paneles en el panel de contenido

La interfaz puede incluir también atributos propios de la aplicación, que no corresponden a compontentes gráficas. Como el resto de los atributos de la clase, se declaran privados y solo pueden ser accedidos por los servicios propios y las clases internas.
Es muy importante estructurar el código de la GUI para favorecer la legibilidad. En una interfaz muy simple los comentarios pueden ser suficientes. Cuando el constructor requiere varias líneas de código es aconsejable utilizar métodos auxiliares
Por supuesto el código puede estructurarse siguiendo un criterio diferente.

- Establecer los valores de los atributos gráficos heredados de la clase *JFrame*.
- Crear los paneles.
- Crear la etiqueta.
- Crear el botón incrementar y el botón decrementar.
- Crear el oyente del botón incrementar y el oyente del botón decrementar.
- Registrar cada uno de los dos oyentes al botón.
- Insertar la etiqueta y los botones en el panel que corresponda.
- Insertar los paneles en el panel de contenido.

Programación basada en Eventos
La construcción de una GUI utiliza un modelo de **programación basado en eventos**. En este modelo el orden en el cual se ejecutan las instrucciones de un programa queda determinado por **eventos**. Un evento es una **señal** de que algo ha ocurrido. Así, el flujo de ejecución no lo establece el programador, sino las acciones del usuario.
Algunas componentes de una GUI son **reactivas**, pueden **percibir** las acciones del usuario y reaccionar en respuesta a ellas. Una componente reactiva está asociada a un **objeto fuente del evento** creado por el programador.
La reacción del sistema en respuesta a la acción del usuario va a quedar determinada por la clase a la que pertenece un **objeto oyente**. El objeto oyente está ligado al objeto fuente de evento a través de una instrucción de **registración**.

Objetos y Eventos
Algunos de las componentes de una GUI son **reactivas**, están asociadas a **objetos fuente de evento** que **perciben eventos externos y disparan eventos internos**. Cuando se dispara un evento interno se crea implícitamente un **objeto evento de software** que pasa como parámetro en un mensaje enviado a un **objeto oyente**.
El objeto oyente se *registra* al objeto fuente de evento asociado a la componente reactiva. En respuesta al mensaje el oyente ejecuta un método que corresponde a la acción del usuario.



Diferentes tipos de componentes requieren implementar diferentes interfaces para manejar los eventos internos que disparan los objetos ligados a las componentes.
Un botón dispara eventos llamados **eventos de acción** que son manejados por **oyentes de acción**. Es decir, un objeto fuente de evento de clase *JButton* detecta un evento externo provocado por la **acción** del usuario sobre el botón, *dispara* un **evento interno** que crea un objeto evento *e* de clase *ActionEvent*. Los objetos de la clase *ActionEvent* requieren implementar la interface *ActionListener*, escribiendo el código del método *actionPerformed* que recibe al objeto evento como parámetro.
La clase de un **objeto fuente de evento** determina así las clases de los **objetos evento** que se crearán implícitamente. La clase del **objeto evento** determina las interfaces de los **objetos oyente** que se deben implementar para establecer el comportamiento asociado a la componente gráfica sobre la que el usuario realizó una acción.
El **objeto oyente** es instancia de una clase que implementa una interface y redefine el método responsable de reaccionar ante la acción del usuario. El **objeto evento** es un parámetro para el método manejador del evento.

Eventos del Mouse
Cuando un usuario interactúa con una interface gráfica realiza acciones que generan eventos de **alto nivel** y de **bajo nivel**. Estos eventos pueden ser percibidos por la aplicación o no.
Si el usuario oprime el mouse sobre un botón y lo suelta sobre el mismo botón, aunque antes de soltarlo lo arrastre fuera de esta componente gráfica, el objeto fuente de evento captura el evento externo y dispara un **evento interno de alto nivel**, esto es, crea un **objeto evento** de clase *ActionEvent.*
Si en cambio el usuario oprime el mouse sobre un botón, lo arrastra y suelta fuera del botón, no se dispara un evento interno de alto nivel y por lo tanto no se crea un objeto evento.
Sin embargo, en ambos casos, se disparan **eventos internos de bajo nivel**, relacionados con el mouse e independientes de la componente gráfica. Estos eventos están asociados a objetos de clase *MouseEvent*, uno creado como consecuencia del click sobre una componente, otro al arrastrar y otro al soltar el botón. Otros eventos de bajo nivel se producen cuando se oprimen teclas o se manipulan las ventanas.

La clase de un **objeto evento** determina las interfaces que se deben implementar para establecer el comportamiento de los **objetos oyentes** asociados a la componente gráfica sobre la que el usuario realizó una acción:

Objeto Evento	Interface de oyente	Manejador
ActionEvent	*ActionListener*	*actionPerformed(ActionEvent)*
ItemEvent	*ItemListener*	*itemStateChanged(ItemEvent)*
MouseEvent	*MouseListener* *MouseMotionListener*	*mousePressed(MouseEvent)* *mouseReleased(MouseEvent)* *mouseEntered(MouseEvent)* *mouseExited(MouseEvent)* *mouseClicked(MouseEvent)*
KeyEvent	*KeyListener*	*keyPressed(KeyEvent)* *keyRealesed(KeyEvent)* *keyTyped(KeyEvent)*

Los atributos del objeto evento brindan información referida a la ubicación del cursor del mouse, cuántos clicks se hicieron, qué botón que se apretó, etc.
La clase del objeto oyente asociada a una componente que detecta acciones del mouse, tiene que implementar los métodos provistos por la interface *MouseListener* o *MouseMotionListener*. El método *addMouseListener* permite registrar el objeto oyente a la componente que va a percibir los eventos del mouse.
Dado que la clase *MouseEvent* hereda de *InputEvent*, sobre un objeto de la clase *MouseEvent* también pueden usarse los métodos definidos en la clase *InputEvent*.

Encapsulamiento y GUI
En el diseño de una aplicación, la solución se modula de modo tal que cada clase pueda implementarse sin depender de las demás. En el desarrollo de una aplicación en la cual la interacción con el usuario se realiza a través de una GUI, la clase que implementa la interface gráfica de usuario usa a las clases que modelan el problema, sin conocer detalles de la representación. Análogamente, las clases que modelan el problema se diseñan e implementan si saber si la entrada y salida va a hacerse por consola o mediante una GUI.

Herencia, Polimorfismo y GUI
Una GUI en Java se construye a partir de objetos gráficos de algunas de las clases gráficas provistas por los paquetes AWT y Swing o de clases derivadas. Conceptualmente la implementación de GUI está fuertemente ligada a los conceptos de herencia, polimorfismo y vinculación dinámica.
Correctitud y GUI
La verificación de una aplicación no garantiza que es correcta pero permite asegurar que funciona de acuerdo a los requerimientos para un conjunto de casos de prueba. Cuando la interacción con el usuario se realiza a través de una GUI el flujo de la ejecución queda determinado por eventos que se generan en respuesta a las acciones del usuario. En este contexto es muy difícil o incluso imposible, anticipar todos los flujos de ejecución posibles. En la verificación se analiza el comportamiento de cada componente considerando diferentes flujos de ejecución.

Productividad y GUI

Una de las razones que hicieron de Java un lenguaje muy utilizado en la industria de software es porque favorece la **reusabilidad** a través del uso de **paquetes** y la **extensibilidad** a través de **herencia**. Cada paquete que se importa se usa como una caja negra, conocimiento únicamente la interface de las clases provistas. Cada clase provista por el lenguaje puede ser utilizada para crear objetos o para definir clases más especializadas, con atributos y servicios específicos de la aplicación

El lenguaje brinda muchos recursos prefabricados que contribuyen a mejorar la productividad, entre ellos una colección de clases gráficas organizadas en forma jerárquica y distribuidas en paquetes. El programador importa solo los paquetes que su aplicación requiere. Cada paquete ofrece una colección de clases que pueden ser usadas para:

- Crear objetos gráficos asociados a las componentes de la interfaz
- Definir clases más específicas a partir de las cuales se crearán componentes.

AWT y **Swing** brindan clases que pueden especializarse para crear botones, cajas de texto, menúes, etc. Una de las ventajas de Swing sobre AWT es que permite desarrollar aplicaciones con una apariencia similar a la de la plataforma subyacente con muy poco esfuerzo. Swing no reemplaza a AWT sino que la usa y agrega nuevas clases.

Las clases de los diferentes paquetes están vinculadas a través de relaciones de **asociación** y **herencia**. Un frame por ejemplo tiene una dimensión y tres botones, entre otras componentes. Las clases *Dimension* y *JButton* están asociadas a *JFrame*. La clase *JButton* a su vez hereda de *JComponent*.