IPOO

El Proceso de Desarrollo de Software

Un programa es una secuencia de instrucciones escritas en un lenguaje de programación, que permiten resolver un problema. Un programa es una componente del software que se desarrolla como parte de un sistema para resolver un problema.

El software está formado por un conjunto de programas y los documentos que modelan el problema y su solución.

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.

La escala de un problema está relacionada con el costo y el tiempo que demanda resolverlo. El proceso se realiza en el marco de un proyecto que establece un cronograma y un presupuesto para la elaboración de un producto. El producto es justamente un sistema de software. 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.

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.

Cualquiera sea el modelo de ciclo de vida, en el proceso intervienen diferentes participantes, entre ellos el cliente que demanda el producto, el equipo de profesionales que lo desarrolla y los usuarios que en definitiva van a interactuar con el sistema.

Etapas

El desarrollo de software es un proceso de que abarca distintas etapas y requiere de la aplicación de una metodología.
Es un proceso colaborativo en el que interactúan los miembros del equipo de desarrollo con clientes y usuarios. Las etapas del proceso de desarrollo puede organizarse de diferentes maneras, una alternativa es el modelo en cascada.

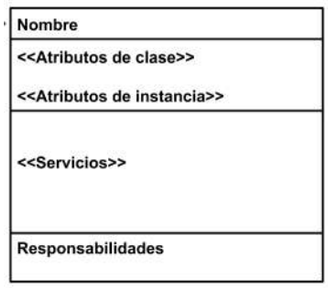
Desarrollo de los Requerimientos

Un sistema de software se desarrolla para resolver un problema que puede surgir de una necesidad, una oportunidad o una idea. 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 especifica:

- 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. Los miembros de una clase son:
•Atributos de instancia y de clase.
•Servicios, pueden ser constructores o métodos.



Implementación

El proceso de desarrollo de software requiere de la aplicación de una metodología y algunas herramientas consistentes con esa metodología.

Una metodología está formada por un conjunto de métodos, técnicas y estrategias. Actualmente la metodología más difundida está integrada al paradigma de programación orientada a objetos. Las herramientas más importantes dentro del proceso de desarrollo de software son el lenguaje de modelado y el lenguaje de programación

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. Es importante que al menos una parte de la verificación la lleven a cabo personas ajenas a la implementación.

Durante el ciclo de vida de un sistema de software las necesidades del usuario cambian y normalmente crecen. El diseño modular es fundamental para controlar el impacto de los cambios. En un sistema bien modulado los cambios menores impactan sobre un conjunto reducido de módulos o incluso pueden llegar a provocar la necesidad de agregar nuevos módulos, sin afectar a los que ya están implementados y verificados.

El paradigma de programación orientada a objetos y los modelos evolutivos son particularmente adecuados para ambientes dinámicos con altos requerimientos de calidad y productividad.

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.

- 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.
Cualquiera sea el modelo de proceso, si una etapa del desarrollo de un producto de software se saltea o no se completa adecuadamente, las etapas siguientes sufrirán las consecuencias, el proyecto probablemente termine demandando más tiempo y el costo global será mayor al presupuestado.

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 si puede
  utilizarse para la construcción de diferentes aplicaciones.

La programación orientada a objetos favorece la reusabilidad y la extensibilidad

Programación orientada a objetos

> 📌 **El modelo computacional de la programación orientada a objetos es un mundo poblado de objetos comunicándose a través de mensajes.**

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.

En la etapa de diseño se establece el comportamiento de los objetos y se completa la especificación de las clases y relaciones. En la actualidad UML es el lenguaje de modelado más utilizado cuando se aplica la programación orientada a objetos. UML permite elaborar diferentes tipos de diagramas, en particular diagramas de clases.

Dos conceptos centrales tanto en la metodología como en las herramientas son objeto y clase.

Una responsabilidad representa un compromiso para la clase o un requerimiento. Las restricciones, requisitos y la funcionalidad de los servicios puede especificarse a través de notas o como un texto que acompaña al diagrama.

Los lenguajes orientados a objetos tienen:

• Encapsulamiento
• Abstracción
• Herencia
• Polimorfismo
Puede tener o no Ligadura dinámica

Objetos y clases

Objetos

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.

Cuando el sistema de software está en ejecución, se crean objetos de software. La ejecución se inicia cuando un objeto recibe un mensaje y en respuesta a él envía mensajes a otros objetos

La palabra objeto se utiliza entonces para referirse a:

- Los objetos del problema, es decir, las entidades identificadas durante el desarrollo de requerimientos o el diseño. 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.
- Los objetos de software, esto es, cada representación que modela en ejecución a una entidad del problema. 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.

Clases

Los objetos del problema pueden agruparse en clases de acuerdo a sus atributos y comportamiento. Todos los objetos de una clase van a estar modelados por un mismo conjunto de atributos y un mismo comportamiento.

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.

En la ejecución del sistema se crean objetos de software, cada uno de ellos es instancia de una clase.

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. Cada objeto brinda el conjunto de servicios que define su clase.

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.

Los atributos se definen a través de la declaración de variables. Un atributo de clase se declara como una variable estática.

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.

Una clase puede brindar varios constructores, siempre que tengan diferente número o tipo de parámetros. Si en una clase no se define explícitamente un constructor, el compilador crea automáticamente uno.

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.

Un sistema de software orientado a objetos está conformado por una colección de clases.

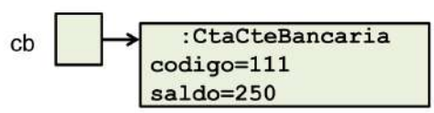
Una clase tester es aquella que verifica los servicios de una o más clases para un conjunto de casos de prueba. Los casos de prueba pueden ser valores:

- Fijos establecidos en el código de la clase tester
- Leídos de un archivo, por consola o a través una interfaz gráfica
- Generados al azar

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.

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 valor de la variable cb es una referencia a un objeto de clase CtaCteBancaria

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.

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, sólo 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.
Cada bloque crea un nuevo ambiente de referenciamiento, formado por todos los identificadores que pueden ser usados. Los operandos de una expresión pueden ser constantes, atributos, variables locales y parámetros visibles en el ambiente de referencia en el que aparece la expresión.

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.

Sintaxis de mensaje en Java

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.

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.

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.

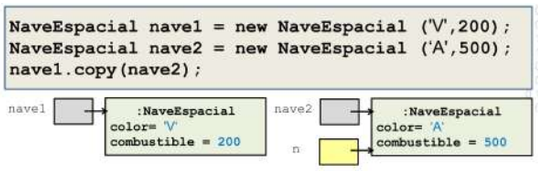
Para comparar el estado interno de los objetos, se comparan los valores de los atributos de instancia.

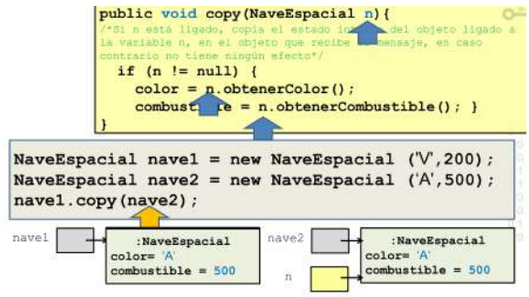
```
public String toString(){
    return codigo+" "+saldo;
}
```

El nombre toString () es estandar para referirse a una consulta que retorna una cadena de caracteres cuyo valor es la concatenación de los valores de los atributos
del objeto que recibe el mensaje.

Copy

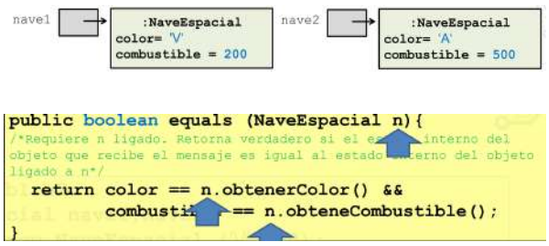
Copia el estado interno del objeto ligado a la variable n

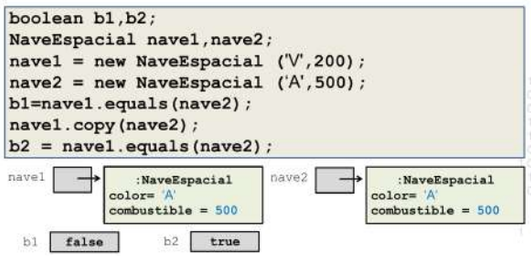




Equals

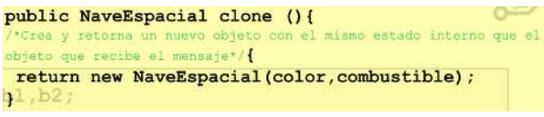
Retorna verdadero si el estado interno del objeto que recibe el mensaje es igual al estado interno del objeto ligado a n

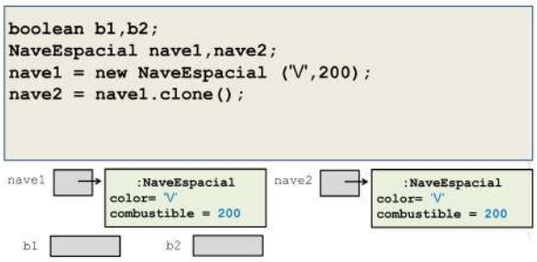




Clone

Crea y retorna un nuevo objeto con el mismo estado interno que el objeto que recibe el mensaje





Memoria

Cada celda de memoria tiene una dirección y un contenido. Tanto la dirección como el contenido se representan mediante 8 bits. 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.

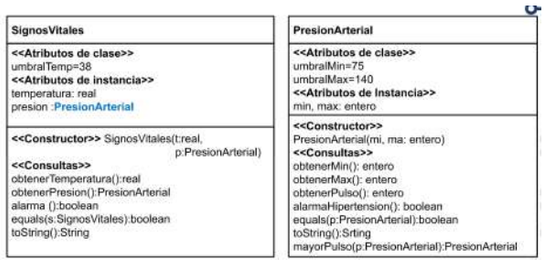
Cuando en ejecución se alcanza una declaración de variable cuyo tipo es una clase, la variable es de tipo clase, se reserva también un bloque de memoria que mantendrá inicialmente una referencia no ligada.

El estado interno del objeto almacena los valores de las variables que corresponden a los atributos de instancia del objeto, determinados por su clase.

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.

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.



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 objetivos 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.**

Representación por referencia

El estado interno de un objeto contiene referencias a los objetos de las clases asociadas, de modo que un sistema complejo puede modelarse a partir de objetos simples. La modificación de una clase, no afecta la representación de los objetos de las clases asociadas.

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, 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.

```
public boolean equals (SignosVitales s){
//Implementa equals superficial. Requiere s ligado
   return temperatura == s.obtenerTemperatura &&
          presión == s.obtenerPresion();
}
```

Igualdad superficial

```
public boolean equals (SignosVitales s){
//Implementa equals en profundidad. Requiere s ligado
   return temperatura == s.obtenerTemperatura &&
          presion.equals(s.obtenerPresion());
}
```

Igualdad en profundidad

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.

```
public Circulo clone (){
//Se implementa en profundidad
  return new Circulo (radio, centro.clone());
}
```

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.

Superficial

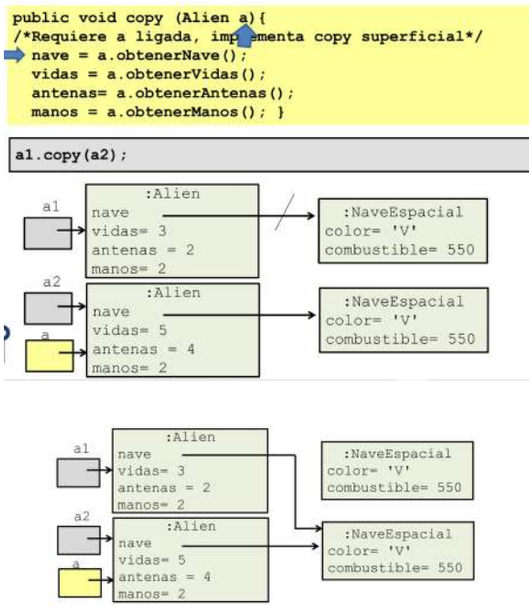
```
public void copy (TermoTanque t ){
//Requiere t ligado
  mechero= t.encendido();
  capacidad = t.obtenerCapacidad();
  termostato =  t.obtenerTermostato();
}
```

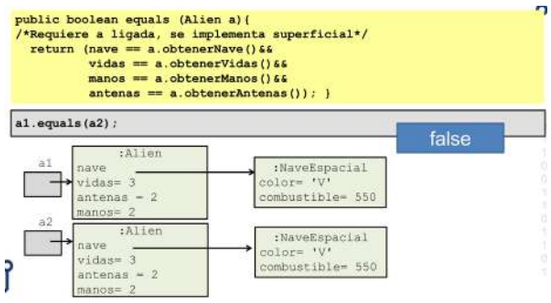
Profundidad

```
public void copy (TermoTanque t ){
//Requiere t ligado
  mechero= t.encendido();
  capacidad = t.obtenerCapacidad();
  termostato.copy(t.obtenerTermostato());
}
```

```
public String toString(){
    return temperatura+"/"+
      presion.toString();
}
```

La consulta toString envía el mensaje toString al objeto ligado al atributo de instancia presion.





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.

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

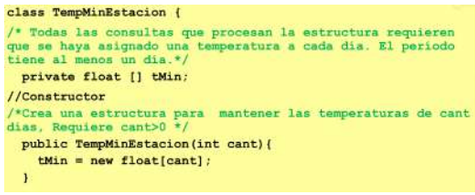
Entre una clase cliente y una clase proveedora de servicios, se establece un contrato que determina las responsabilidades de cada una. Las condiciones del contrato se especifican en la etapa de diseño del sistema e incluyen la funcionalidad y los requisitos inherentes a cada servicio. En la implementación 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.

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.

Encapsulamiento y Abstracción

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.



```
class TempMinEstacion {
/* Todas las consultas que procesan la estructura requieren
que se haya asignado una temperatura a cada dia. El periodo
tiene al menos un dia.*/
  private float [] tMin;
//Constructor
/*Crea una estructura para  mantener las temperaturas de cant
dias, Requiere cant>0 */
  public TempMinEstacion(int cant){
    tMin = new float[cant];
  }
```

```
public int cantDias(){
    return tMin.length;
}
```

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. Cuando surgen cambios en la especificación o en el diseño, el encapsulamiento permite reducir el impacto. 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.

Para que un objeto de software oculte su estado interno, la clase que define sus atributos y comportamiento debe esconder la implementación.

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.

Encapsulamiento y oscurecimiento de información son dos conceptos relacionados pero diferentes. El primero agrupa atributos y servicios, el segundo esconde la implementación.

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.

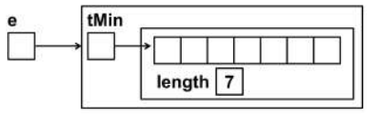
Estructura de datos

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.

Los atributos se declaran privados y quedan así escondidos dentro de la clase.

```
class TestTempMinEstacion{
  public static void main(String[] args){
  // Tester para una semana
    int cant =7;
    TempMinEstacion est = new TempMinEstacion(cant);
```



Representaciones alternativas para estructuras lineales y homogéneas

Se utilizan 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.

Patrones de diseño de algoritmos

Con frecuencia los recorridos sobre las estructuras de datos responden a patrones de algoritmos, independientes del tipo de los elementos de la estructura. Por ejemplo, es habitual recorrer una estructura de datos lineal y homogénea para contar la cantidad de elementos que satisfacen una propiedad, el algoritmo puede ser entonces:

```
Algoritmo contar
  Dato de Salida: contador
  contador se inicializa en 0
  para cada elemento de la estructura
    si el elemento cumple la propiedad
    incrementar el contador
```

Los algoritmos que implementan los servicios provistos por una clase están escondidos dentro de la clase. Si se modifican, en tanto no cambie la signatura, la funcionalidad ni los requisitos, la modificación es transparente para las clases cliente.

Otro recorrido habitual sobre una estructura consiste en hallar el mayor elemento, de acuerdo a una relación de orden establecida. En este caso el algoritmo es:

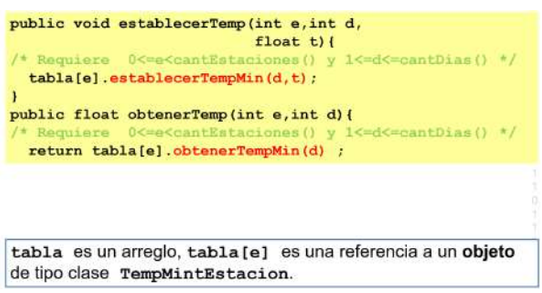
```
Algoritmo mayor Dato de salida:  mayor
  mayor := primer elemento
  para cada elemento de la estructura a partir del segundo
  si elemento > mayor
    mayor := elemento
```

Para decidir si algún elemento de la estructura verifica una propiedad no es necesario hacer un recorrido exhaustivo:

```
Algoritmo verifica Dato de salida: verifica
verifica := falso
para cada elemento de la estructura y mientras no verifica
  si el elemento actual verifica la propiedad
    verifica := verdadero
```

Encapsulamiento y clases relacionadas

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

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.

El diseño de una estructura de datos está fuertemente ligado a cómo van a accederse las componentes. Los arreglos son adecuados para representar estructuras de datos homogéneas y lineales, cuyas componentes deben ser accedidas directamente a través de su posición.

Una clase que encapsula de forma parcialmente ocupada ocurre cuando el constructor crea una estructura para mantener cierta cantidad máxima de componentes. Desde la clase cliente, se insertan y eliminan componentes de acuerdo a funcionalidad especificada en el diseño. Toda la entrada y salida se hace desde la clase cliente de la clase que encapsula a la estructura de datos.

El término colección hace referencia 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.

```
//Comandos
public void insertar (SignosVitales s) {
/*Asigna s a la primer posiciones libre de la colección y aumenta la
cantidad de posiciones ligadas. Requiere que s esté ligado y la
clase cliente haya verificado que la colección no esté llena.*/
  T[cant++] = s;
}
```

```
//Consultas
public int cantAlarmas(){
//Retorna la cantidad de mediciones con alarma
  int contador=0;
  for (int i=0;i<cant;i++)
    if (T[i].alarma())
      contador++;
  return contador;
}
```

```
//Consultas
public int cantMediciones(){
//Retorna la cantidad de mediciones registradas
  return cant;
}

public boolean estaLlena(){
//Retorna true si la colección está llena
  return cant==T.length;
}
```

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 una estructura que encapsula un arreglo que mantiene referencias nulas y ligadas intercaladas y brinda operaciones para insertar, eliminar, buscar y procesar de alguna manera los elementos.

```
class Sectores {
   private Androide[] T;
//Constructor
public Sectores(int max) {
/*Crea una tabla para representar max sectores */
    T= new Androide [max];
}
```

```
class Estacionamiento {
   private Micro[] T;
//Constructor
public Estacionamiento(int max) {
/*Crea una tabla para representar max unidades de
estacionamiento */
    T= new Micro [max];
}
```

```
public int cantUnidadesOcupadas (){
/*Retorna la cantidad de unidades ocupadas por un
micro*/
  int i = 0; int cant = 0;
  while (i < cantUnidades ()){
    if (T[i]!=null) cant++;
      i++;  }
  return cant;
}
```

```
public boolean estaAndroide(Androide a) {
/*Retorna true si el androide a está asignado al
menos a un sector*/
  boolean esta=false;
  for (int i=0;i < cantSectores() && !esta;i++)
    esta = T[i] == a;
  return esta;
}
```

```
public void retirar(int p) {
/*Elimina el micro de la unidad p. Requiere
controlada la unidad*/
  T[p] = null;
}
```

```
public boolean existeUnidad (int p){
   return p>= 0 & p< T.length;
}
```

```
public Micro microUnidad (int p){
/*Retorna el micro estacionado en la unidad p.
Require p válida */
  return T[p];
}
```

```
Algoritmo seleccion
repertir para i tomando valores entre la primera posicion y la anterior a la ultima
    buscar la posicion pos del menor elemento desde la posicion i
    intercambiar los elementos dde las posiciones i y pos
```

```
// Compara cada elemento de la estructura con el siguiente y los intercambia si corresponde
// El proceso se repite hasta que la estructura está ordenada en su totalidad.
void bubbleSort(int arr[]){
    for (int i = 0; i < arr.length-2; i++)
        // Last i elements are already in place
            if (arr[i] > arr[i + 1])
                intercambiar(arr[i], arr[i + 1]);
}
void intercambiar(int xp, int yp){
    int aux = xp;
    xp = yp;
```

```
    yp = aux;
}
```

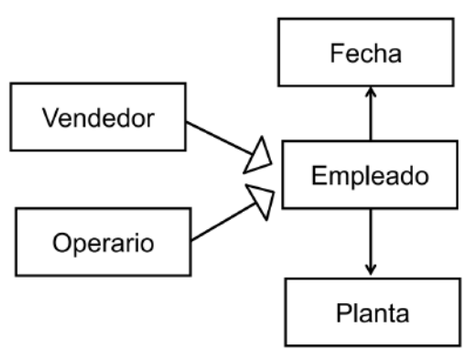
Cuando una colección de elementos está ordenada de acuerdo a un atributo y es necesario decidir si un elemento en particular pertenece a la colección, es posible aplicar una estrategia conocida como búsqueda binaria. La estrategia consiste en partir la estructura en mitades, considerando que el elemento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

```
Algoritmo Busqueda Binaria(Elemento e){
/* Decide si el elemento e se encuentra almacenado en la
        colección.*/
        int inicio, fin, mitad;
        boolean encontre=false;
        inicio=0;
        fin=cant-1;
        while (inicio<=fin && !encontre){
            mitad=(inicio+fin)/2;
            if (f[mitad].equals(e))
            encontre=true;
            else if (f[mitad].esMayor(e))
            fin = mitad-1;
            else
            inicio = mitad+1;
        }
            return encontre;
}
```

Herencia y polimorfismo

La herencia jerárquica es un mecanismo que permite organizar clases de acuerdo a relaciones de generalización-especialización. 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. 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.



Las clases Vendedor y Operario son clases derivadas de la clase base Empleado

Todo objeto de clase Vendedor es un objeto de clase Empleado.

Todo objeto de clase Operario es un objeto de clase Empleado.

El estado interno de cada instancia de Operario tiene los atributos específicos de su clase y los heredados de la clase Empleado.

Las clases derivadas de la clase base Empleado heredan atributos y métodos, no los constructores.

```
class Empleado {
  protected int legajo;
  protected float sueldoBasico;
  protected Fecha fechaIngreso;
   //Constructor




   //Comandos

   //Consultas

}
```

Los atributos protegidos pueden ser accedidos desde las clases derivadas

```
class Operario extends Empleado  {
   //Atributos de instancia
   protected int horas;
   //Constructor
   public Operario (int leg, float sb, Fecha fi){
     super(leg,sb,fi);
   }
   //Comandos
   public void establecerHoras(int h){
     horas = h; }
   //Consultas
  public int obtenerHoras (){
     return horas; }
   public float obtenerPremio (float p){
     return horas * p ; }
}
```

SUPER llama al constructor de la clase padre

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.

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.

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

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 redefinición se produce cuando una clase derivada define un método con la misma signatura que un método de una clase ancestro. El método de la clase padre queda derogado para las instancias de la clase hija.

Cuando dos o más métodos tienen el mismo nombre pero distinto tipo o número de parámetros el identificador está sobrecargado.

Polimorfismo

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.

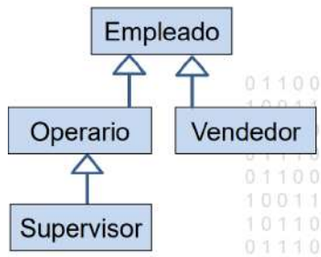
El polimorfismo permite que las diferentes entidades de una misma clase puedan exhibir distintas formas de un mismo comportamiento. En la programación orientada a objetos el polimorfismo es el mecanismo que permite que un mismo nombre pueda quedar ligado a objetos de diferentes clases y que objetos de distintas clases puedan recibir un mismo mensaje y cada uno actúe de acuerdo al comportamiento establecido por su clase.

- 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.

- Un método polimórfico incluye una o más variables polimórficas como parámetro.

Empleado e1, e2, e3, e4;
Las variables e1, e2, e3 y e4 son polimórficas, pueden estar ligadas a objetos de clase Empleado o cualquiera de sus clases derivadas.

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.

```
Empleado e1, e2, e3, e4;
e1 = new Operario(125,21000,f);
e2 = new Supervisor(126,20000,f);
e3 = new Vendedor(120,30000,f,100000);
e4 = e2;
Operario e5 = new Supervisor (129,32000,f);
Supervisor e6 = new Supervisor(128,30000,f);
```

Variable	Tipo Estático	Tipo Dinámico
e1	Empleado	Operario
e2	Empleado	Supervisor
e3	Empleado	Vendedor
e4	Empleado	Supervisor
e5	Operario	Supervisor
e6	Supervisor	Supervisor

En Java se utiliza 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.

Ambiente de referenciamiento

El ambiente de referenciamiento de una clase es el conjunto de nombres que son visibles y pueden ser utilizados en toda la clase, incluye:

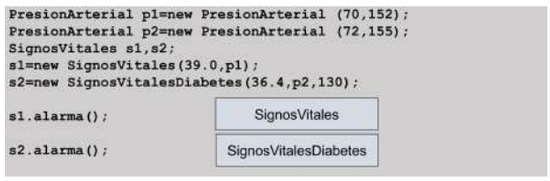
• Los atributos de instancia y de clase definidos en la clase.
• Los servicios definidos en la clase.
• Todos atributos y servicios públicos de las clases relacionadas por asociación y dependencia.
• Cualquier otra clase definida en el mismo proyecto.
• Las clases públicas definidas en los paquetes importados.
• Todos los atributos y servicios protegidos de las clases ancestro.

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

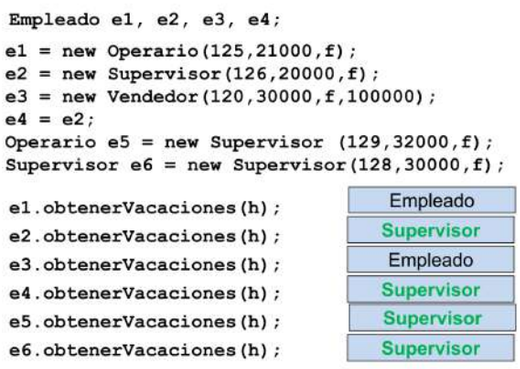
clase derivada, el tipo dinámico de la variable determina qué método va a ejecutarse en respuesta a un mensaje.

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.



La ligadura es dinámica, la clase del objeto determina qué método se ejecuta en respuesta a un mensaje.

Las ligaduras estáticas o vinculaciones estáticas son las que se establecen antes de la ejecución.
Las ligaduras dinámicas o vinculaciones dinámicas son las que se establecen y pueden cambiar durante la ejecución.



Cuando un objeto recibe un mensaje Java busca un método en la clase que corresponde al tipo dinámico, si no existe, busca en la clase
padre y así siguiendo en la rama de ancestros hasta llegar a Object.

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

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. El casting con tipos elementales convierte, con TDA no.

```
public boolean equals (Empleado e) {
/*Implementa igualdad en profundidad, requiere e ligado y fecha de
ingreso ligada*/
 boolean igual=false;
 if (this == e) igual = true;
 else if (this.getClass() == e.getClass())
   igual = legajo == e.obtenerLegajo() &&
           sueldoBasico == e.obtenerSueldoBasico() &&
           fechaIngreso.equals(e.obtenerFechaIngreso);
 return igual;
}
```

En la clase Empleado

```
public boolean equals (Empleado e) {
  boolean igual=false;
  if (this == e) igual = true;
  else if (this.getClass() == e.getClass()){
   Gerente g = (Gerente) e;
   igual = super.equals(e) && productividad==g.obtenerProductividad();}
  return igual;
}
```

En la clase Gerente

El casting le “indica” al compilador que no debe realizar el chequeo de tipos, el programador asegura que el tipo dinámico de e es Gerente, es decir e mantiene una referencia a una instancia de Gerente.

Si se ejecuta el método equals de la clase Gerente, podemos asegurarle al compilador que el objeto que recibió el mensaje es de clase Gerente y también que el parámetro es de la misma clase.
La asignación es válida porque el casting relaja el control del compilador.

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.

```
abstract class Contenedor {
//atributos de  instancia
protected float volumen;
protected float densidad;
//constructor
public Contenedor(float v,float d){
  volumen = v; densidad = d;}
//consultas
public float obtenerVolumen(){
 return volumen;}
public float obtenerDensidad(){
 return densidad;}
abstract public float obtenerImpacto();
}
```

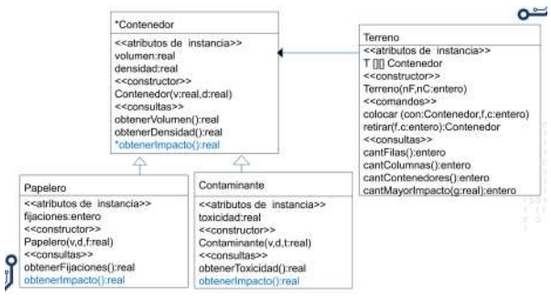
Si una clase hereda de una clase abstracta y no implementa todos los métodos abstractos, también debe ser definida como abstracta. El constructor de una clase abstracta sólo va a ser invocado desde los constructores de las clases derivadas. Una clase concreta debe implementar todos los métodos abstractos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

Tipos estáticos y dinámicos

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.

```
public float mayorToxicidad(){
 float mayor=0; Contaminante c;
 for (int i = 0; i< cantFilas();i++)
   for (int j = 0; j< cantColumnas();j++)
     if (T[i][j] != null){
         c = (Contaminante) T[i][j];
         if (c.obtenerToxicidad() > mayor)
           mayor = c.obtenerToxicidad();
       }
 return mayor;
}
```

El compilador no reporta error pero la ejecución terminará anormalmente si el terreno contiene referencias a contenedores que no son contaminantes.




```
PresionArterial p1=new PresionArterial (70,152);
PresionArterial p2=new PresionArterial (72,155);
SignosVitales s1,s2;
s1=new SignosVitales(39.0,p1);

SignosVitalesDiabetes d1,d2;

d1=new SignosVitales(39.0,p1);   Error

d2=s1;                           Error
```

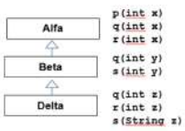
El chequeo de tipos de Java es estático, el tipo de la
variable determina a qué objetos puede quedar ligada.

```
Empleado o1 = new Operario(125,21000,f);
Operario o2,o3;

o2 = o1;
o3 = new Empleado(126,20000,f);
```

Errores de compilación

```
Alfa v1 = new Alfa();
Alfa v2 = new Beta();
Alfa v3 = new Delta();
```

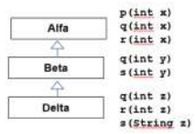



```
v1.q(1);   Alfa
v2.q(1);   Beta
v3.q(1);   Delta
```

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 dinámico determina la ligadura entre el
mensaje y el método.

```
Alfa v1 = new Beta();
Beta v2 = new Delta();
Beta v3 = new Delta();
```




```
v1.s(1);          Error
v2.s(1);
v3.s("abc");      Error
```

Generalización

La generalización es un proceso que consiste en abstraer lo que es común y esencial en un conjunto de entidades, para formar un concepto general que comprenda a todas. Es decir, la generalización es una forma de abstracción que mediante la cual las propiedades comunes de instancias específicas se utilizan para crear conceptos generales.

La generalización es la base esencial de todas las inferencias deductivas válidas en las ciencias.

En Ciencias de la Computación una función, procedimiento o método puede pensarse como una generalización que describe el flujo de control a partir del cual se mapea un valor o conjunto de valores en otro valor o conjunto de valores.

La genericidad es un paso más en el proceso de generalización. La genericidad aumenta las oportunidades de reuso, porque evita escribir el mismo código repetidamente.

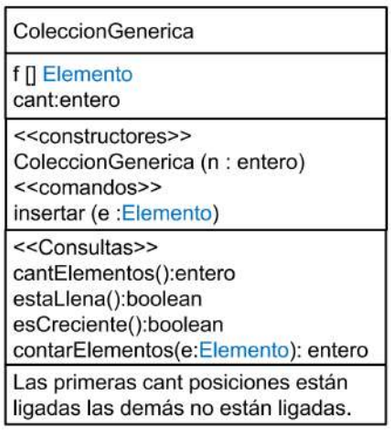
Una función genérica se implementa independientemente del tipo de los parámetros. Una clase genérica encapsula a una estructura cuyo comportamiento es independiente del tipo de las componentes.

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:

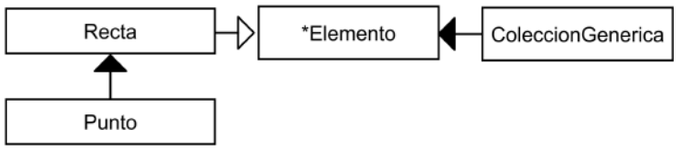
- En la etapa de diseño reducir la complejidad en base a la descomposición del problema en piezas más simples, a partir de un conjunto de clases relacionadas entre sí.
- En la etapa de implementación utilizar un lenguaje que permita retener las relaciones entre las clases y encapsular su representación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

Colección generica

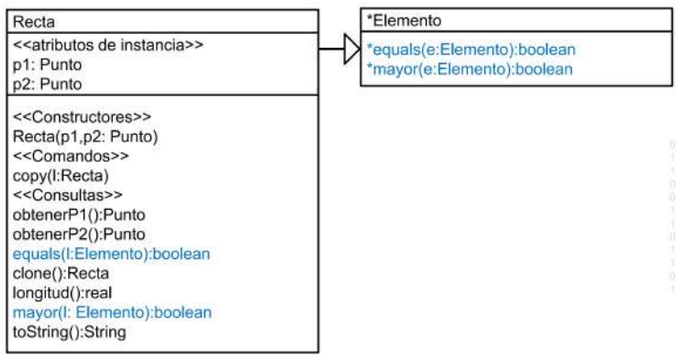


La clase ColeccionGenerica encapsula a un arreglo f cuyas componentes son de tipo Elemento

```
public boolean equals(Elemento l){
/*Requiere l ligado, se implementa
superficial*/
  Recta r = (Recta) l;
  return ((p1==r.obtenerP1()) &&
          (p2==r.obtenerP2()));
}
```

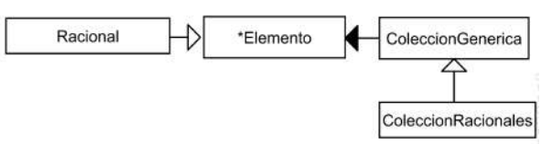


De acuerdo a este ESTE diagrama una objeto de clase ColeccionGenerica va a mantener referencias a objetos de clase Recta



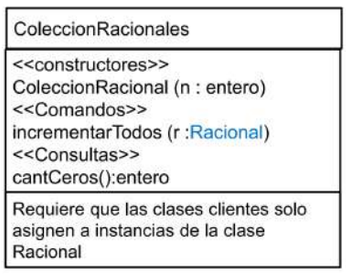
Si Recta extiende a Elemento este diseño sobrecarga los métodos equals y mayor, no los redefine.

Una clase genérica encapsula a una estructura cuyo comportamiento es independiente del tipo de sus elementos. La genericidad puede modelarse usando herencia o polimorfismo paramétrico.



ColeccionRacionales extiende a ColeccionGenerica
y agrega métodos específicos

```
public int cantCeros (){
/* Cuenta de racionales que representan a un 0*/
  int cont =0;
  for (int i=0;i<cantElementos();i++)
   if (((Racional)f[i]).obtenerNum() == 0)
     cont++;
  return cont;
}
```




```
public void incrementarTodos (Racional r){
/* Incrementa cada elemento de la colección en r.
Requiere r ligado*/
  for (int i=0;i<cantElementos();i++)
   ((Racional)f[i]).suma(r) ;
}
```

Colecciones ordenadas

Aunque un objeto de clase ColeccionOrdenadaGenerica pueden contener componentes que son instancias de clases que extienden a Elemento, en una aplicación particular todas las componentes son instancias del mismo tipo.

```
Algoritmo Insertar Ordenado
public void insertar(Elemento e){
        if (e!=null){
            int pos= buscarPosInsercion(e);
            arrastrarElementos(pos);
            t[pos]=e;
            cant++;
        }
    }
    protected int buscarPosInsercion(Elemento e){
        /* Retorna la posicion del primer elemento de la
        colección mayor a e. Si todos son menores retorna
        cant.*/
        int pos= 0;
        boolean salir=false;
        while (pos<cant && !salir){
            if (t[pos].esMayor(e))
            salir=true;
            else
            pos++;
        }
        return pos;
    }
    protected void arrastrarElementos(int pos){
        /* A partir de pos, arrastra todos los elementos una
        posición hacia adelante.*/
        int posAux=cant;
        while (posAux>pos){
            t[posAux]=t[posAux-1];
            posAux--;
        }
    }
```

```
Algoritmo Eliminar Ordenado(elemento e)
public void eliminar(Elemento e){
        /* Elimina en la colección el elemento equivalente a e */
        int pos;
        if (e!=null){
            pos= buscarPosElemento(e);
            if (pos<cant) {
                arrastrarElementosAnt(pos);
                cant--;
            }
        }
    }
    protected int buscarPosElemento (Elemento e){
        /* Retorna la posicion de la primera aparición del elemento e. Si
        no existe retorna cant.*/
```

```
        int pos= cant;
        boolean encontre=false;
        for (int i=0; i<cant && !encontre && !t[i].esMayor(e); i++){
            if (t[i].equals(e)){
                encontre=true;
                pos=i;
            }
        }
        return pos;
    }
    protected void arrastrarElementosAnt(int pos){
        /* A partir de la posición siguiente a pos, arrastrar todos los
        elementos una posición hacia atrás.*/
        while (pos<cant-1){
            t[pos]=t[pos+1];
            pos++;
        }
        t[pos]=null;
    }
```

Tabla genérica

En lugar de utilizar una clase abstracta elemento se crea la clase tabla genérica utilizando como clase de la tabla, la clase Object

```
class TablaGenerica {
   protected Object[] T;
//Constructor
public TablaGenerica(int max) {
/*Crea una tabla para representar max posiciones. Requiere
max mayor a 0 */
    T= new Object [max];
}
```

```
class Sectores extends TablaGenerica {
//Constructor
public Sectores(int max) {
/*Crea una tabla para representar max posiciones.
Requiere max mayor a 0 */
    super (max);
}
```

```
public boolean estaObjeto(Object o) {
/*Retorna true si el objeto o está ligado a una posición*/
  boolean esta=false;
  for (int i=0;i < cantPosiciones() && !esta;i++)
    esta = T[i] == o;
  return esta;
}
```

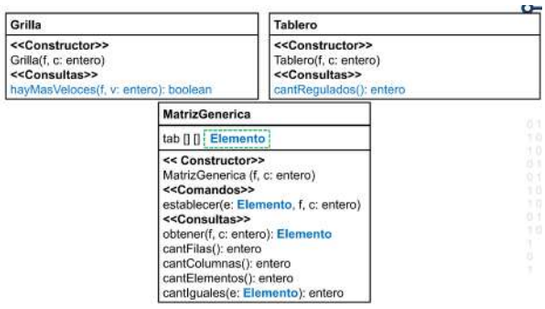
Los servicios puede generalizarse porque no dependen del tipo de las componentes de la tabla

```
…
Androide a = new Androide (20,n);
Sectores e = new Sectores(2000);
e.asignar(a);
```

```
//Comandos
public void asignar (Object o) {
/*Asigna o a la primera posición no ligada.*/
  int i = 0;
  while (T[i] != null && i < cantPosiciones())
    i++;
  if (i < cantPosiciones())
    T[i] = o;
}
```

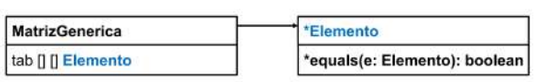
El parámetro real a se asigna al parámetro formal o, es una asignación polimórfica

Matriz genérica



Las consultas cantRegulados y hayMasVeloces son específicas de cada aplicación y no pueden generalizarse.

La clase genérica MatrizGenérica está asociada a la clase abstracta Elemento.



La funcionalidad y las responsabilidades son independientes del tipo de las componentes

Interfaz gráfica de usuario

Una interface gráfica de usuario (GUI) es un medio que permite que una persona se comunique y controle un sistema a través de ventanas, botones, menúes, etc.

Una GUI se construye a partir de una colección de componentes con una representación gráfica y capacidad para percibir eventos generados por las acciones del usuario.

- interfaz: medio de comunicación entre entidades.
- gráfica: incluye ventanas, menúes, botones, texto, imágenes.
- usuario: persona que usa la interfaz para controlar un sistema

Algunas componentes tienen la capacidad para percibir eventos generados por las acciones del usuario y reaccionar en respuesta a ese evento. La creación de componentes reactivas requiere que el lenguaje brinde algún mecanismo para el manejo de eventos.

La construcción de GUI está fuertemente relacionada con los conceptos de:

• Abstracción
• Encapsulamiento
• Herencia
• Polimorfismo
GUI y POO
Y a la productividad:
• Reusabilidad
• Extensibilidad

Implementación de una GUI

• Crear un objeto gráfico para cada componente de la GUI e insertarlo en otras componentes contenedoras.
• Establecer los valores de los atributos de los objetos gráficos.

• Definir el comportamiento de las componentes reactivas en respuesta a las acciones del usuario

Un objeto gráfico es una instancia de una clase gráfica. Una clase gráfica define algunos atributos gráficos brinda servicios gráficos.
Una clase gráfica puede usarse para:

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

Una interface es una colección de constantes y signaturas de métodos que en cada aplicación pueden implementarse de manera específica.

Una clase interna es una clase que se declara como miembro de otra clase y solo puede ser utilizada para crear objetos dentro de la clase en la que está declarada.

Paquetes gráficos

Un paquete agrupa a una colección de clases relacionadas entre sí.

Usamos clases gráficas provistas por los paquetes AWT Abstract Window Toolkit) o Swing o derivadas de una ellas. AWT y Swing son entonces paquetes que facilitan la construcción de interfaces gráficas. Ambos brindan una colección de clases que permiten crear botones, cajas de texto, menúes, etc.

Si vamos a usar la clases provistas por Swing, ya sea para extenderlas o para crear objetos, debemos importar el paquete.

Swing es una extensión del paquete AWT y brinda clases que permiten enriquecer la apariencia, accesibilidad y usabilidad de las GUI.

Los tipos de componentes pueden agruparse en:

- controles básicos
- displays interactivos con información altamente formateada
- displays con información no editable
- contenedores

La clase JComponent, derivada de Container, es la clase base de Swing, generaliza el comportamiento de la mayoría de las demás clases de este paquete.

Cada clase derivada de JComponent modela un tipo de componente y tiene atributos propios.
Cada tipo de componente tiene una apariencia estándar pero también puede configurarse de acuerdo a las pautas de diseño que se adopten, estableciendo los valores de los atributos.

Una ventana es una componente contenedora. Algunos de sus atributos son el borde, la barra de título y el panel de contenido sobre el que se insertan las componentes.

Un frame es un tipo especial de ventana sobre el que se ejecuta una aplicación. En Java la clase JFrame permite crear un frame o una clase que la especialice. Toda instancia de JFrame tiene atributos marco, barra de título, algunos botones y un panel de contenido. La clase JFrame brinda servicios para modificar los valores de los atributos. Un objeto de clase JFrame contiene a otras componentes gráficas: barra de título, borde, tres botones y un panel de contenido.

Una etiqueta es un objeto gráfico pasivo que permite mostrar un texto y/o una imagen. En Java podemos crear una etiqueta definiendo un objeto de clase JLabel. Una etiqueta tiene atributos tamaño, texto, imagen, alineación de la imagen y alineación del texto.

Un botón es un objeto gráfico reactivo cuyo comportamiento podemos especificar registrándo el botón. En Java podemos crear una etiqueta definiendo un objeto de clase JButtom. Un botón tiene entre sus atributos tamaño, texto, imagen y puede estar habilitado o deshabilitado.

También es posible agrupar botones de modo que solo uno esté seleccionado.
Cada botón es una instancia de la clase JRadioButton. El conjunto de botones se insertan en un objeto de clase ButtonGroup.

```
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()
```

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.

```
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 comportamiento de las componentes reactivas de la GUI va a quedar definido en las clases internas.

Un objeto de la clase JTextField permite leer o mostrar un campo de texto y debe registrarse a un oyente para que pueda detectar y reaccionar ante un evento externo.

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

Un panel es un área sobre la que trabaja el usuario o se colocan otras componentes. Los paneles pueden como contenedores de otros paneles u otro tipo de componentes. Para definir un panel creamos un objeto de la clase JPanel. El principal atributo de un panel es el organizador de diagramado que permite especificar como se distribuyen las componentes en su interior.

Los paneles van a quedar organizados de manera jerárquica. Esto es, el panel de contenido contiene paneles que a su vez pueden contener a otros paneles. El panel de contenido va a tener su organizador de diagramado y para cada uno de los paneles en que se divide podemos establecer también un organizador de diagramado.

La distribución de componentes en paneles facilita el diseño de GUI. Por lo general el diseñador organiza la componentes de un frame en paneles. 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:

- FlowLayout: Distribuye los componentes uno al lado del otro comenzando en la parte superior.
- 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.

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 estandar usamos la clase JOptionPane. Todo diálogo es dependiente de un frame. Cuando un frame se destruye, también se destruyen los diálogos que dependen de él.

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 entrada y salida 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.

Estructura general de una GUI

- Importar paquetes
- Declarar los objetos gráficos
- Establecer los valores de los atributos del frame
- Crear los objetos gráficos
- Establecer el diagramado de los objetos gráficos contenedores
- Establecer los valores de otros atributos de los objetos gráficos
- Crear los oyentes
- Registrar los oyentes a los objetos gráficos reactivos
- Insertar los objetos gráficos en los contenedores
- Implementar las interfaces que determinan el comportamiento de los oyentes.

Programación basada en eventos

La programación basada en eventos es un modelo de programación en el cual el orden de la ejecución lo determina las acciones del usuario.

Algunas componentes de una GUI son reactivas, pueden percibir las acciones del usuario y reaccionar en respuesta a ellas.

Cuando un usuario realiza una acción sobre una componente reactiva se genera un evento. Un evento es una señal de que algo ha ocurrido.

Consideraremos únicamente eventos generados por acciones del usuario al interactuar con la GUI.

Los objetos fuente del evento, están asociados a una componente de la interfaz, tienen una representación gráfica y son capaces de percibir y reaccionar ante un evento externo provocado por una acción del usuario y disparar eventos de software.

Los objetos evento que son disparados implícitamente por un objeto fuente del evento.

Los objetos oyentes (listeners) se ejecutan para manejar un evento. La clase a la que pertenece un objeto oyente brinda métodos para manejar eventos, es decir especifica el curso de acción a seguir en respuesta a diferentes tipos de eventos.

Ejemplos en GUI

El método add es polimórfico, puede recibir como parámetro a una etiqueta, un botón u otro tipo de componentes gráficas siempre y cuando tenga un ancestro común.

El método addActionListener es polimórfico, recibe como parámetro a un objeto de una clase que Java no conoce.

```
/*Insertar los botones en el panel de contenido*/
  getContentPane().add(botonRojo);
  getContentPane().add(botonVerde);
```

```
/*Crear y registrar los oyentes para los botones*/
  OyenteBotonR ponerRojo = new OyenteBotonR();
  botonRojo.addActionListener(ponerRojo);
  OyenteBotonV ponerVerde = new OyenteBotonV();
  botonVerde.addActionListener(ponerVerde);
```

```
import java.awt.*;
import javax.swing.*;
class MiVentana extends JFrame{

    public MiVentana() {
      super("Mi Segunda GUI");
      setSize(400,200);
      getContentPane().setBackground(Color.BLUE);
}
}
```

Obtiene el atributo **panel de contenido** del frame y le envía un mensaje para establecer el color con la constante BLUE de la clase estática Color.

La clase JFrame brinda servicios que permiten establecer los valores de los atributos del frame, por ejemplo el diagramado y el tamaño.

`setSize(100, 120);`

Establece el tamaño del frame.

`JLabel etiqueta= new JLabel("Hola!");`

Crea una etiqueta estableciendo su texto.

`getContentPane().add(etiqueta);`

Recupera el panel del contenido del frame e inserta en su interior la etiqueta.

`setDefaultCloseOperation(EXIT_ON_CLOSE);`

Establece terminar la aplicación cuando el usuario cierre la ventana.

```
/*Crear y registrar los oyentes para los botones*/
  OyenteBotonR ponerRojo = new OyenteBotonR();
  botonRojo.addActionListener(ponerRojo);
  OyenteBotonV ponerVerde = new OyenteBotonV();
  botonVerde.addActionListener(ponerVerde);
```

Para que un botón sea reactivo debe registrarse a un objeto oyente.

```
/*definir clases para establecer el comportamiento de
  los botones */
class OyenteBotonR implements ActionListener {
   public void actionPerformed (ActionEvent event){
     getContentPane().setBackground(Color.red);
   }
}
class OyenteBotonV implements ActionListener {
     public void actionPerformed(ActionEvent event){
      getContentPane().setBackground(Color.green);
    }
}
```

Para que un botón reaccione ante una acción del usuario, debe estar registrado a un oyente de una clase que implementa a la interface ActionListener.

```
private void armaPanelEtiqueta(){
  panelImagen = new JPanel();
  etiqueta = new JLabel();
  etiqueta.setIcon(new ImageIcon("manzana.gif"));
  panelImagen.add(etiqueta);
  getContentPane().add(panelImagen);
}
```

- Crea un panel como un objeto de clase **JPanel**
- Crea una etiqueta como un objeto de clase **JLabel**
- Establece la imagen de la etiqueta con el contenido del archivo manzana.gif
- Inserta la etiqueta en el panel
- Inserta el panel en el panel de contenido