CIMAT

Centro de Investigación en Matemáticas, A.C.

Principios de Diseño aplicados a Árboles Binarios de Búsqueda

REPORTE TÉCNICO

Que para obtener el grado de

Maestro en Ingeniería de Software

P r e s e n t a

I. S. C. Elizabeth Acuña Cháirez

Director de Reporte Técnico

Dr. Jorge Roberto Manjarrez Sánchez

Zacatecas, Zacatecas. 20 de agosto de 2012

Agradecimientos

Durante estos 2 últimos años de estudio, mi vida ha atravesado por buenos y malos momentos que me ayudaron a fortalecer mi carácter y me brindaron una perspectiva de la vida mucho más amplia. Al finalizar mis estudios de ***maestría en ingeniería de software*** *existen un grupo de personas a las que no puedo dejar de reconocer debido a que durante todo este tiempo estuvieron presentes de una u otra forma evitando que me perdiera en el proceso.*

A ***Dante Ramos Orozco****...mi esposo, gracias por estar en los momentos más importantes de mi vida, siempre apoyándome y dándome ánimos para no dejarme vencer y cada día me ayudas a seguir creciendo personal y profesionalmente.*

A ***mi familia,*** *le agradezco por apoyarme siempre, por darme ánimos y por mostrarse orgullosos de mí y de mis capacidades. Gracias a ustedes sé que no estoy sola y que cuento con personas que me quieren y me respaldan siempre.*

A ***mis profesores****, por sus formas tan diferentes de enseñar y sus virtudes, me incentivaron en muchos sentidos a seguir adelante en mi preparación.*

A ***mis compañeros****, algunos se volvieron mis amigos, de los que aprendí muchas cosas, buenas y malas, gracias a ustedes tuve una visión más objetiva y general de lo que es el ámbito profesional y ahora me siento más preparada para asumir cualquier reto.*

A las empresas e instituciones que me apoyaron*:*

- ***CIMAT****, por servir de enlace entre mis aspiraciones y mis logros.*
- ***COZCyT*** *(Consejo Zacatecano de Ciencia y Tecnología), por el financiamiento económico que recibí durante el periodo de estudios de maestría y principalmente para la preparación de este reporte.*

Ing. Elizabeth Acuña Cháirez
estudiante CIMAT
elaicz@hotmail.com

Resumen. Dentro del desarrollo del software suelen presentarse varios problemas causados por un diseño degradado, entre los que se pueden mencionar la rigidez, la fragilidad y la inmovilidad del software, pero al corregir el diseño, éstos se corrigen junto con él. En este trabajo se presentan los Principios Fundamentales de Diseño, los cuales son reglas o normas que nos orientan como ingenieros de software en la creación de un buen diseño que ayude a evitar problemas en la creación de módulos, estructuración de clases, realización de pruebas y en el mantenimiento de una aplicación.

A lo largo de este trabajo se ilustra el uso de los principios con ejemplos de código en Java y diagramas en UML. Se aplicarán los principios de diseño a la creación de una librería de árboles binarios de búsqueda y se demostrará su función en el diseño de la misma.

Palabras clave. Diseño de software, componentes, modularidad, herencia, interfaz, diagramas UML, árboles binarios.

Abstract. Within software development usually several problems are caused by a rotting design, some of these problems include rigidity, fragility and immobility of the software, if the design is corrected the problems also disappear. This work presents the Fundamental Principles of Design; these principles are rules or standards that guide us as software engineers to create a strong design that helps to avoid problems in the creation of modules, structuring classes, and testing and maintenance of an application.

Throughout this work, usage of the principles is illustrated with examples in Java code along with their corresponding UML diagrams. Furthermore, the design principles are used to design a library of binary search trees to illustrate their application in the design.

Keywords. Software design, components, modularity, inheritance, interface, UML diagrams, binary trees.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Índice de Diagramas

1 Introducción

"El software es el equipamiento lógico o soporte lógico de un sistema informático, comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas, en contraposición a los componentes físicos, que son llamados hardware." [1]

Para crear software hacemos uso de la arquitectura de software que es: "la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución." [2]

La arquitectura de software es multinivel, como se ilustra en la Figura 1, en el nivel más alto encontramos patrones de arquitectura que definen la forma y estructura de las aplicaciones de software, esto es la visión estática, que describe qué componentes tiene la arquitectura.

En el nivel siguiente esta la arquitectura que esta específicamente relacionada con el propósito de la aplicación de software, también conocida como la visión funcional, que describe qué hace cada componente.

En el nivel más bajo reside la arquitectura de los módulos y sus interconexiones, llamada visión dinámica, donde se describe el comportamiento de los componentes a lo largo del tiempo y su interacción entre sí. Esto es el dominio de los patrones de diseño, paquetes, componentes y clases. Y es aquí donde empiezan a surgir las degradaciones en el diseño, que causan que los programas de software fallen.

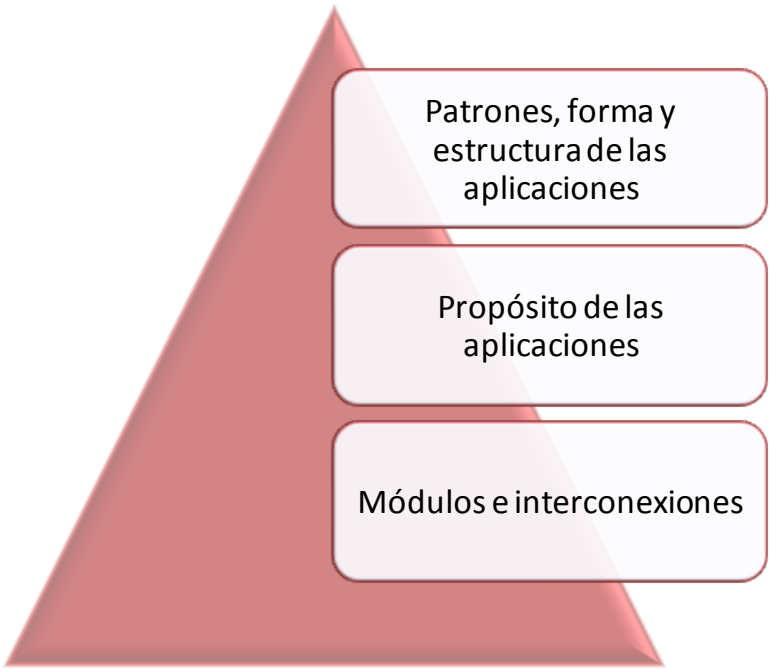

Figura 1. Niveles de la arquitectura de software

1.1 Síntomas de un Diseño Degradado

El software tiende a fallar principalmente por errores insertados durante las fases de diseño y codificación, los errores insertados en codificación principalmente son errores de sintaxis y son fáciles de encontrar gracias a las múltiples herramientas auto corregibles que existen, mientras que los errores insertados en el diseño son errores de lógica y son más difíciles de detectar. Robert Martin identificó 4 tipos de síntomas que indican cuando un diseño esta degradado [3]:

1. La **rigidez**, es la tendencia que sufre el software a ser difícil de cambiar. Esto debido a que cada cambio causa una cascada de cambios subsecuentes en módulos dependientes.

2. La **fragilidad**, es la predisposición que sufre el software a fallar en muchos lugares cada vez que es cambiado. A menudo el fallo ocurre en áreas que no tienen relaciones conceptuales con el área que fue cambiada.

3. La **inmovilidad**, es la falta de habilidad para reutilizar el software de otros proyectos o partes del mismo proyecto. Esto ocurre a menudo cuando un ingeniero se da cuenta que necesita un modulo que es similar a uno que un ingeniero ya escribió. Sin embargo, a menudo sucede que el modulo en cuestión tiene mucho equipaje que depende de él. Después de mucho trabajo los ingenieros descubren que el trabajo y el riesgo que se requiere para separar partes de software deseables de las partes indeseables son demasiados. Por lo que se vuelve más fácil escribir el software que reutilizarlo.

4. La **viscosidad**, es la propiedad que impide mantener el diseño después de realizar un cambio, o realizar un cambio dado el diseño que se tiene. Viene en dos formas: viscosidad del diseño y la viscosidad del ambiente. En la viscosidad de diseño, cuando nos enfrentamos a un cambio, usualmente existe más de una manera de hacer dicho cambio, algunas de las maneras preservan el diseño, otras no. La viscosidad del ambiente se da cuando se tiene un ambiente de desarrollo ineficiente para realizar dichos cambios.

Además de estos 4 síntomas, en la literatura se mencionan otros 3 síntomas [4]:

5. La **complejidad innecesaria**, es la tendencia a agregar infraestructura en el diseño que no proporciona beneficios. Esto es un problema porque se anticipan cambios que luego no se producen, lo que conduce a más código (que hay que depurar y mantener). La complejidad innecesaria hace que el software sea más difícil de entender.

6. La **repetición innecesaria**, es la característica que se produce por "Copiar y pegar" sin analizar el funcionamiento del código y ver si es requerido. El mantenimiento se vuelve difícil. El código duplicado debería unificarse bajo una única abstracción.

7. Y por último la **opacidad**, es la tendencia que sufre el código al volverse difícil de entender. La cantidad de información que contiene el código tiende a aumentar si no se toman las medidas necesarias.

Cada uno de los 7 síntomas mencionados anteriormente es causado directa o indirectamente por dependencias inapropiadas entre los módulos de software, dichas dependencias son establecidas en base a la lógica que debe seguir el programa.

Es la arquitectura de dependencias la que se degrada y con ella la capacidad del software de ser mantenido. Con el propósito de evitar esta degradación, las dependencias entre los módulos deben ser administradas, dicha administración consiste en la creación de cortafuegos de dependencias, estos cortafuegos son una parte del sistema que bloquea el acceso no autorizado, permitiendo al mismo tiempo comunicaciones autorizadas. A través de los cortafuegos las dependencias no se propagan.

Dentro del Diseño Orientado a Objetos tenemos principios que apoyan en la construcción de dichos cortafuegos. Dentro de este trabajo nos enfocaremos en los 10 Principios Fundamentales de Diseño popularizados por Robert Cecil Martin en el año 2000, estos principios son reglas propuestas que deben seguirse para eliminar el diseño degradado, y fueron publicados a través de Object Mentor [3].

1.2 Descripción del Contenido

Dentro del contenido desarrollado en este trabajo encontramos las siguientes áreas:

- **Marco Teórico**. Donde se revisan conceptos básicos de diseño, principios propuestos por otros autores y un análisis de la relación de dichos principios con los síntomas del diseño degradado.

- **Desarrollo**. Se presentan los 10 principios propuestos por Robert C. Martin, para el diseño de software. Se describe el nivel más alto de desarrollo que han tenido estos principios, hasta el momento. Y se aplican dichos principios al diseño de una librería de árboles binarios de búsqueda, utilizando diagramado UML y código de programación en Java.

- **Conclusiones**.- Descripción de experiencias durante la aplicación de los principios.

- **Trabajo Futuro**. Agregados al estudio presentado.

- **Referencias.** Fuentes de información consultadas para la elaboración de este trabajo.

- **Apéndices.** Información complementaria de los temas presentados a lo largo del trabajo.

2 Marco Teórico

En este capítulo se explican los conceptos necesarios para comprender el contexto dentro del cual se desarrollan los Principios Fundamentales de Diseño popularizados por Robert C. Martin.

2.1 Conceptos de Diseño

Abstracción.

Es la descripción del sistema donde se resaltan algunos detalles y se suprimen otros. Denota las características esenciales que distinguen a un objeto de otros tipos de objetos, definiendo precisas fronteras conceptuales, referentes al observador.

Deben seguir el "principio de mínimo compromiso", que significa que la interface de un objeto provee su comportamiento esencial, y nada más que eso. Pero también el "principio de mínimo asombro": capturar el comportamiento sin ofrecer sorpresas o efectos laterales.

Encapsulación.

Es la ocultación de detalles de un objeto que no contribuyen a sus características esenciales. La encapsulación sirve para separar la interface de una abstracción y su implementación.

La encapsulación da lugar a que las clases se dividan en dos partes:

- Interface: captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
- Implementación: comprende la representación de la abstracción, así como los mecanismos que conducen al comportamiento deseado.

Los conceptos de abstracción y encapsulación son conceptos complementarios: abstracción hace referencia al comportamiento observable de un objeto, mientras encapsulación hace referencia a la implementación que la hace alcanzar este comportamiento.

Jerarquización.

Es el orden de relación que se produce entre abstracciones diferentes. Los tipos de jerarquía más útiles:

- Herencia (generalización/especialización, padre/hijo, jerarquía del tipo "es un"). Una clase (subclase) comparte la estructura o comportamiento definido en otra clase, llamada superclase.
- Herencia múltiple. Una clase comparte la estructura o comportamiento de varias superclases.
- Agregación. Comprende relaciones del tipo "es parte de" al realizar una descomposición.
- Relaciones, entre los conceptos asociados al modelo de objetos.

Existe una tensión entre los conceptos de encapsulación de la información y el concepto de jerarquía de herencia, que requiere una apertura en el acceso a la información. C++ y Java ofrecen mucha flexibilidad, pudiendo disponer de tres compartimentos en cada clase:

- Privado: declaraciones accesibles sólo a la clase (completamente encapsulado)
- Protegido: declaraciones accesibles a la clase y a sus subclases.
- Público: declaraciones accesibles a todos los clientes.

Modularidad.

Es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos adherentes y remotamente semejantes.

Cada módulo se puede compilar separadamente, aunque tengan conexiones con otros módulos. En un diseño estructural, modularización comprende el agrupamiento significativo de subprogramas. En diseño orientado a objetos, la modularización debe ceñirse a la estructura lógica elegida en el proceso de diseño.

Dividir un programa en componentes individualizados reduce de alguna manera su complejidad.

Estructura de Datos. [5]

Es una forma de organizar los datos. Muestra la manera lógica y congruente como se relacionan los datos. En programación las estructuras de datos representan agrupaciones complejas de datos simples, como lo son las pilas, las colas, los arboles y las listas; pero por sobre todas estas la forma de organización de datos más usadas por los sistemas son las bases de datos, aunque no sean consideradas en sí mismo una estructura de datos.

Representan:

- La organización.
- Los métodos de acceso.
- El procesamiento de la información.
- La capacidad de asociación.

Ejemplos: Escalar, Vector, Pilas, Colas, Listas, Árboles, etc.

Tipificado.

Es la imposición de una clase a un objeto, de tal modo que objetos de diferentes tipos no se puedan intercambiar, o se puedan intercambiar solo de forma restringida. Tipo y clase pueden considerarse sinónimos.

Concurrencia.

Es la propiedad que distingue un objeto activo de uno no activo. La concurrencia permite que diferentes objetos actúen al mismo tiempo, usando distintos hilos de control.

Persistencia.

Es la propiedad por la cual la existencia de un objeto trasciende en el tiempo (esto es, el objeto sigue existiendo después de que su creador deja de existir) o en el espacio (esto es, la localización del objeto cambia respecto a la dirección en la que fue creado). [6]

Procedimiento de Software.

Es la descripción detallada de pasos que corresponde a las funciones de cada uno de los módulos del software, pudiendo ser representado a través de algoritmos y diagramas de flujo.

División o Partición Estructural.

Es la segmentación en la arquitectura de software, que puede ser divida en forma horizontal y vertical, como se ilustra en la Figura 2. Al dividirla de forma horizontal podemos ver que cada sección representará una función del software y al dividirla de forma vertical podremos observar que cada sección nos muestra un nivel de trabajo de el software, mostrando en la parte superior los módulos que toman decisiones y en la parte inferior los módulos que realizan transacciones.

a) Horizontal: F1, F2, F3 (E, P, S)
- Fácil prueba y mantenimiento
- Poca propagación efectos secundarios
- Software fácilmente ampliable.

b) Vertical: Descomposición en factores
- TOP -> Control
- DOWN -> Procesamiento
- Menos susceptibles a efectos secundarios

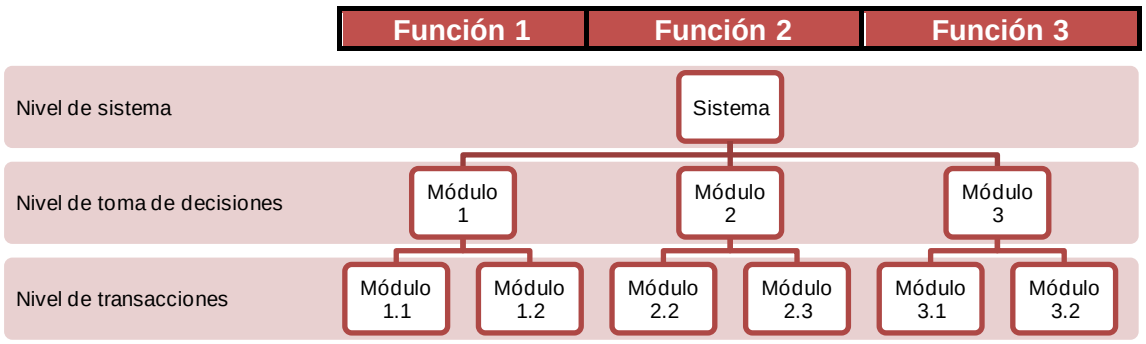

Figura 2. División estructural de diseño

2.2 Otros Principios de Diseño

En esta sección se presentan principios de diseño propuestos por otros autores además de Robert Martin, dichos principios solo son tomados como referencia del trabajo que ya se ha realizado, y no serán presentados más adelante en el documento.

2.2.1 Principios de Diseño, Alan M. Davis (1995)

Los principios de diseño propuestos en su obra "201 principios de desarrollo de software [7]":

1. En el proceso deben tomarse enfoques alternativos.
2. Deberá rastrearse hasta el análisis.
3. Se debe reutilizar.
4. Tratar de imitar el dominio del problema.
5. Uniformidad e integración.
6. Deberá estructurase para admitir cambios.
7. Debe prever la adaptación a circunstancias inusuales.
8. No codificar.
9. Evaluarse en función de calidad mientras está creciendo.
10. Minimizar errores conceptuales.

2.2.2 Principios para el Diseño de Datos, Wasserman (1996)

Los principios de diseño de datos propuestos en su obra "Hacia una disciplina de la Ingeniería del Software [8]":

1. Los principios sistemáticos del análisis aplicado a la función y al comportamiento también deben aplicarse a los datos.
2. Deben identificarse todas las estructuras de datos y las operaciones que se han de realizar sobre cada una de ellas.
3. Debe establecerse y usarse un diccionario de datos para definir el diseño de los datos y del programa.
4. Deben posponerse las decisiones de datos de bajo nivel hasta el diseño detallado.
5. La representación de una estructura de datos sólo debe ser conocida por los módulos que hagan uso directo de los datos contenidos en la estructura.
6. Se debe desarrollar una biblioteca de estructuras de datos útiles y de las operaciones que se les pueden aplicar.
7. El diseño del software y el lenguaje de programación deben soportar la especificación y la realización de tipos abstractos de datos.

2.2.3 Principios del Diseño de Software, Pressman (2006)

Los principios de diseño propuestos en su obra "Ingeniería de Software: Un Enfoque Práctico [9]":

1. En el proceso de diseño no deberán utilizarse "orejeras" (no estar aislado de los demás procesos).
2. El diseño deberá poderse rastrear hasta el modelo de análisis.
3. El diseño no deberá inventar nada que ya esté inventado.
4. El diseño deberá minimizar la distancia intelectual entre el software y el problema, como si de misma vida real se tratara.
5. El diseño deberá presentar uniformidad e integración.
6. El diseño deberá estructurarse para admitir cambios.
7. El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes.
8. El diseño no es escribir código y escribir código no es diseñar.
9. El diseño deberá evaluarse en función de la calidad mientras que se va creando, no después de terminarlo.
10. El diseño deberá revisarse para minimizar los errores conceptuales (semánticos).

2.2.4 Análisis de Principios Previos

De los principios propuestos por Davis, Wasserman y Pressman, se realizó un análisis para detectar que síntoma de diseño degradado se ataca al aplicar cada principio, se resume dicho análisis en la Tabla 1.

Autor	Principio	Síntoma que ataca
Davis	En el proceso deben tomarse enfoques alternativos.	Viscosidad en el diseño
	Deberá rastrearse hasta el análisis.	Viscosidad en el diseño
	Se debe reutilizar.	Complejidad y Repetición innecesaria
	Tratar de imitar el dominio del problema.	Opacidad
	Uniformidad e integración.	Rigidez
	Deberá estructurase para admitir cambios.	Rigidez
	Debe prever la adaptación a circunstancias inusuales.	Fragilidad
	No codificar.	Complejidad innecesaria
	Evaluarse en función de calidad mientras está creciendo.	Opacidad
	Minimizar errores conceptuales.	Fragilidad

Wasserman	Los principios sistemáticos del análisis aplicado a la función y al comportamiento también deben aplicarse a los datos.	Rigidez
	Deben identificarse todas las estructuras de datos y las operaciones que se han de realizar sobre cada una de ellas.	Rigidez
	Debe establecerse y usarse un diccionario de datos para definir el diseño de los datos y del programa.	Opacidad
	Deben posponerse las decisiones de datos de bajo nivel hasta el diseño detallado.	Fragilidad
	La representación de una estructura de datos sólo debe ser conocida por los módulos que hagan uso directo de los datos contenidos en la estructura.	Inmovilidad
	Se debe desarrollar una biblioteca de estructuras de datos útiles y de las operaciones que se les pueden aplicar.	Inmovilidad
	El diseño del software y el lenguaje de programación deben soportar la especificación y la realización de tipos abstractos de datos.	Rigidez
Pressman	En el proceso de diseño no deberán utilizarse “orejeras” (no estar aislado de los demás procesos).	Viscosidad en el diseño
	El diseño deberá poderse rastrear hasta el modelo de análisis.	Viscosidad en el diseño
	El diseño no deberá inventar nada que ya esté inventado.	Complejidad y Repetición innecesaria
	El diseño deberá minimizar la distancia intelectual entre el software y el problema.	Opacidad
	El diseño deberá presentar uniformidad e integración.	Rigidez
	El diseño deberá estructurarse para admitir cambios.	Rigidez
	El diseño deberá estructurarse para degradarse poco a poco.	Fragilidad
	El diseño no es escribir código y escribir código no es diseñar.	Complejidad innecesaria
	El diseño deberá evaluarse en función de la calidad mientras que se va creando, no después de terminarlo.	Opacidad
	El diseño deberá revisarse para minimizar los errores conceptuales (semánticos).	Fragilidad

Tabla 1. Análisis de principios de diseño previos

3 Principios del Diseño de Software

Este capítulo analiza e ilustra las construcciones de software llamadas principios, en los que se da su definición, un ejemplo de diseño que **no** se ajusta al principio, otro que **si** se ajusta al principio y finalmente un ejemplo codificado en java del principio.

Para la ejemplificación de los principios se hace uso de UML (Unified Modeling Language), específicamente de diagramas de clases, debido a que son los que ayudan a la representación de las clases y sus relaciones, en la Tabla 2 se resume el uso de los diferentes diagramas que proporciona UML, el área, la vista y los conceptos que abarca cada uno.

Área	Vista	Diagramas	Conceptos Principales
Estructural	Vista Estática	Diagrama de Clases	Clase, asociación, generalización, dependencia, realización, interfaz
	Vista de Casos de Uso	Diagramas de Casos de Uso	Caso de Uso, Actor, asociación, extensión, generalización.
	Vista de Implementación	Diagramas de Componentes	Componente, interfaz, dependencia, realización.
	Vista de Despliegue	Diagramas de Despliegue	Nodo, componente, dependencia, localización.
Dinámica	Vista de Estados de máquina	Diagramas de Estados	Estado, evento, transición, acción.
	Vista de actividad	Diagramas de Actividad	Estado, actividad, transición, determinación, división, unión.
	Vista de interacción	Diagramas de Secuencia	Interacción, objeto, mensaje, activación.
Administración o Gestión de modelo	Vista de Gestión de modelo	Diagramas de Clases	Paquete, subsistema, modelo.
Extensión de UML	Todas	Todos	Restricción, estereotipo, valores, etiquetados

Tabla 2. Diagramas de UML por área de aplicación

3.1 El Principio Abierto Cerrado

Un módulo de software debería estar abierto para la extensión, pero cerrado para su modificación. [3], [10]

Principio atribuido a Bertrand Meyer. El Principio Abierto Cerrado (OCP-Open Closed Principle), se enfoca en crear ensamblados, clases y métodos que puedan ser modificados sin grandes complicaciones. Se recomienda evitar la edición del código de una clase que funciona correctamente para llevar a cabo una modificación, es decir, cada vez que se requiera una modificación debemos mejorar el comportamiento de la clase añadiendo nuevo código sin tocar nada de lo que hay desarrollado.

Traducido en buenas prácticas, usar composición y herencia como herramienta para llevar a cabo las modificaciones, la forma más común de cumplir con este principio es **implementar una interfaz en todas las clases que son susceptibles de ser modificadas en un futuro.**

A continuación en el Diagrama 1, se muestra como **no** se aplica el principio abierto cerrado, dado que el método CalcularArea tiene relación directa a la clase Figura, dando a entender que el calcular el área es igual para todas las figuras, lo cual resulta incorrecto.

Diagrama 1. Diseño que no se ajusta al principio abierto cerrado

Para aplicar el principio se requiere crear una interfaz a la que haga referencia el métpodo CalcularArea, como se ilustra en el Diagrama 2. Con esto cada figura tendrá su propio método para calcular su área.

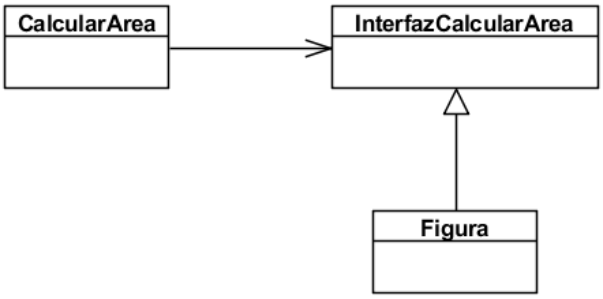

Diagrama 2. Diseño que si se ajusta al principio abierto cerrado

Ejemplificando el principio con código.

Se pide una clase "Geometría" con un método "calcularArea" (que calcula el área de una figura), la primera aproximación seria:

```
public class Geometria {     ...
    public double calcularArea(Figura figura) {
        double ret = 0;
        if (figura instanceof Cuadrado) {
            ret = ((Cuadrado)figura).getLado()
                    * ((Cuadrado)figura).getLado();
        } else if (figura instanceof Rectangulo) {
            ret = ((Rectangulo)figura).getLado1()
                    * ((Rectangulo)figura).getLado2();
        } else if (figura instanceof Triangulo) {
            ret = (((Triangulo)figura).getBase()
                    * ((Triangulo)figura).getAltura()) / 2;
        } if (figura instanceof ...
        return ret;
    }...
  }
```

Con esta aproximación tenemos un grave problema, y es que por cada nueva figura que añadamos a nuestro catálogo, tendremos que modificar el método “calcularArea”. Para poder cumplir con el Principio Abierto Cerrado y solucionar el problema propuesto se pueden realizar diferentes diseños. En este caso vamos a optar por una sencilla solución en la que la responsabilidad de calcular el área de una figura recaerá en sí misma:

```
public interface Figura {
    ....
    double calculaArea();
    ....
}
public class Cuadrado implements Figura {
    ...
    public double calculaArea() {
        return this.getLado() + this.getLado();
    }
    ...
}
public class Rectangulo implements Figura {
    ...
    public double calculaArea() {
        return this.getLado1() + this.getLado2();
    }
    ...
}
public class Triangulo implements Figura {
    ...
    public double calculaArea() {
        return (this.getBase() * this.getAltura()) / 2;
    }    ...
}
public class Geometria {
    ...
    public double calcularArea(Figura figura) {
        return figura.calculaArea();
    }...
}
```

Como se puede observar, si añadimos nuevas clases que implementen la interface no será necesario modificar el código de las clases que ya han sido desarrolladas, por lo que estaremos cumpliendo con el Principio Abierto Cerrado.

3.2 El Principio de Sustitución de Liskov

Subclases deben ser sustituibles por sus clases base. [3], [11]

Este principio fue creado por Barbara Liskov en el año 1987. En el Principio de Sustitución de Liskov (LSP-Liskov Substitution Principle), un usuario de una clase base debe continuar funcionando correctamente si se le pasa una instancia de una clase extendida. El principal objetivo de este principio es asegurar que en la herencia entre clases de la Programación Orientada a Objetos una clase derivada no únicamente **sea** sino que debe se **comporte** como la clase base.

Este principio está estrechamente relacionado con la programación por contratos. Para poder llevar a cabo la sustitución, el contrato de la clase base debe ser honrado por la clase extendida.

- Precondiciones menos fuertes que las de los métodos de la clase base.
- Postcondiciones menos débiles que las de los métodos de la clase base.

En Diagrama 3, aunque se está aplicando herencia entre las clase figura no se está respetando el comportamiento de la misma, ya que las subclases que heredan de ella deben tener el mismo comportamiento. El método dibujar viola el principio de sustitución de Liskov, por que ha de ser consciente de cualquier subtipo de Figura que creemos en nuestro sistema.

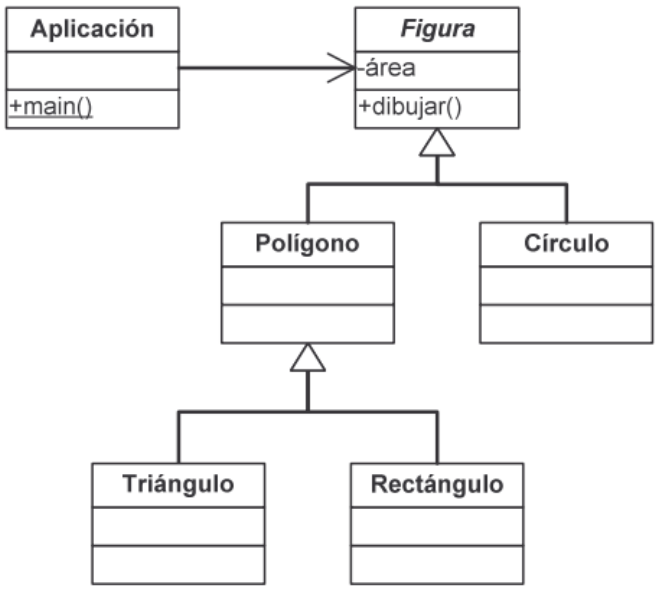

Diagrama 3. Diseño que no se ajusta al principio de sustitución de Liskov

La solución a lo anterior es implementar polimorfismo, cada tipo de figura será responsable de implementar el método dibujar como se aprecia en la Diagrama 4.

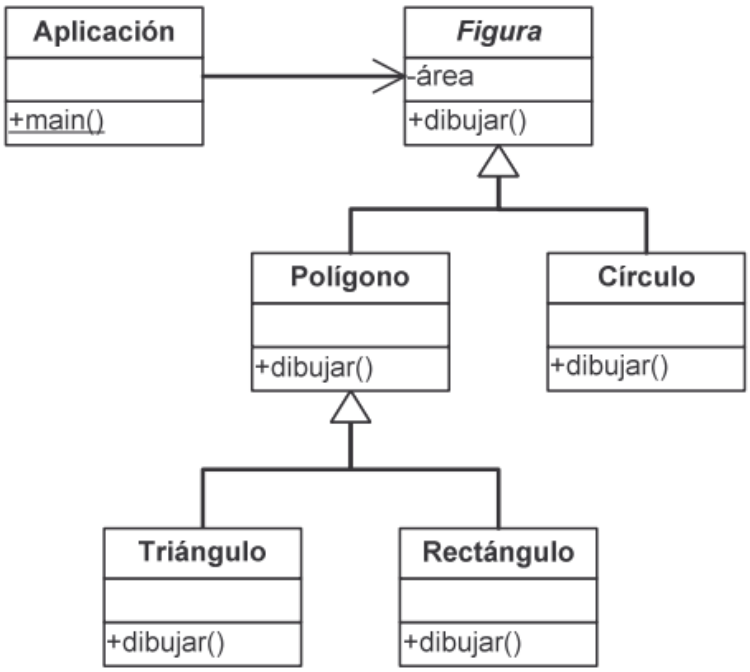

Diagrama 4. Diseño que si se ajusta al principio de sustitución de Liskov

Aplicando el principio al código [12].

```
public class Figura
{
   ...
    public void dibujar();
   ...
}

public class Polígono extends Figura
{
  public void dibujar()
}


public class Circulo extends Figura
{
  public void dibujar()
}


public class Polígono extends Figura
{
  public void dibujar()
}
```

3.3 El Principio de Inversión de Dependencia

Dependerá de abstracciones. No dependerá de implementaciones. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. [3], [13]

Principio definido por Robert C. Martin. En el Principio de Inversión de Dependencia (DIP-Dependency Inversion Principle), los módulos de alto nivel tratan con las políticas de alto nivel de la aplicación. A estas políticas generalmente no le interesan los detalles de sus implementaciones.

Las abstracciones son puntos de articulación; representan los lugares donde el diseño puede doblar o ser extendido, sin modificarse ellas mismas. Cualquier cosa concreta es volátil, la no volatilidad no es un reemplazo para la capacidad de sustitución de una interface abstracta.

La inversión de dependencias se puede realizar siempre que una clase envía un mensaje a otra y deseamos eliminar la dependencia. Como se ilustra en el Diagrama 5, la clase Botón depende de la Clase Lámpara por lo tanto no será posible utilizar el botón para encender, por ejemplo, un motor (ya que el botón depende directamente de la lámpara que enciende y apaga) [14]. Para solucionar el problema, volvemos a hacer uso de nuestra capacidad de abstracción, como se ilustra en el Diagrama 6, para que el Botón elimine la dependencia que tiene de la Lámpara.

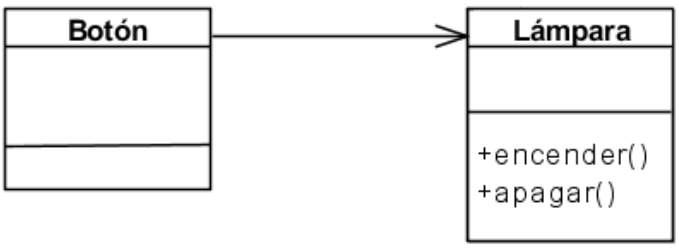

Diagrama 5. Ejemplo de diseño que no se ajusta al principio de inversión de dependencia

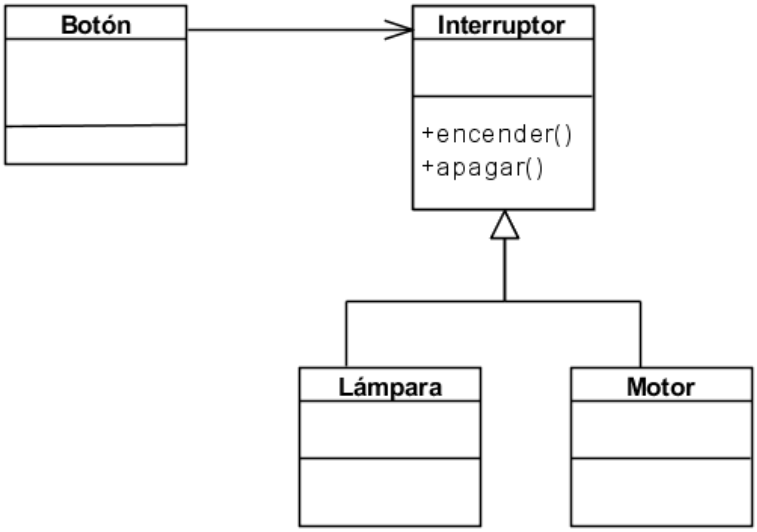

Diagrama 6. Ejemplo de diseño que si se ajusta al principio de inversión de dependencia

Otro ejemplo sería el siguiente, un copiador que cuenta con un Lector de Teclado y un Escritor Disco, donde dada la dependencia ni el lector ni el escritor pueden funcionar sin el copiador, como se ilustra en el Diagrama 4.

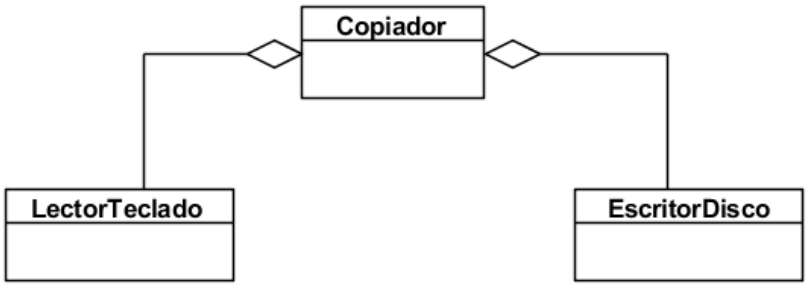

Diagrama 7. Diseño que no se ajusta al principio de inversión de dependencia

En código:

```
public class Copiador
{
  public void copiar(LectorTeclado l, EscritorDeisco e)
  {
    while(!l.eof)
    {
      byte b=l.leer();
      e.escribir(b);
    }
  }
}
```

Para cumplir con el principio de inversión de dependencia, eliminamos la dependencia directa que se tiene con el copiador como se ilustra en el Diagrama 8.

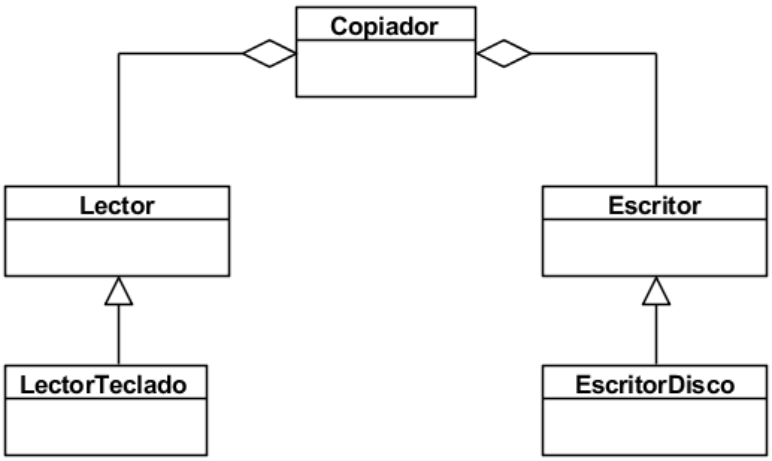

Diagrama 8. Diseño que si se ajusta al principio de inversión de dependencia

```
public class Copiador
{
  public void copiar(Lector l, Escritor e)
  {
    while(!l.eof)
    {
      byte b=l.leer();
      e.escribir(b);
    }
  }
}
```

3.4 El Principio de Segregación de la Interfaz

Muchas interfaces específicas de los clientes son mejor que una sola interfaz de propósito general. Los clientes de una clase no deberían depender de interfaces que no utilizan. [3], [15]

Este principio fue formulado por Robert C. Martin. En el Principio de Segregación de la Interfaz (ISP-Interface Segregation Principle), los clientes deben ser categorizados por su tipo y se deben crear interfaces para cada tipo.

Cuando las aplicaciones orientadas a objetos se mantienen, las interfaces a las clases existentes y componentes generalmente cambian. Este impacto puede ser mitigado

agregando nuevas interfaces a objetos existentes en vez de cambiar las interfaces que ya existen [16].

Si tenemos que hacer un programa que envíe un mail a un contacto con información de un pedido, podemos pensar que la clase EnviarMail en el método enviar acepte el contacto como parámetro.

```
public class Contacto
{
    public string Nombre { get; set; }
    public string CuentaBancaria { get; set; }
    public string Email { get; set; }
}

public interface IMailSender
{
    void Enviar(Contacto contacto, string cuerpoMensaje);
}
```

Ahora bien, ¿Hace falta pasar todo el contacto cuando solo usará la dirección de mail y el nombre? Las interfaces de la clase se pueden dividir en pequeños grupos funcionales. Cada grupo sirve a un cliente diferente. De esta manera, unos clientes conoce una interfaz con un grupo de funcionalidad y otro cliente conoce otro grupo.

A continuación tenemos la interfaz IReceptorMail que solo tiene el nombre y la dirección de correo electrónico. Luego se agrega la interfaz sobre la clase contacto y cambiamos el método de enviar para que use la interfaz IReceptorMail en lugar de la clase contacto.

```
public interface IReceptorMail
{
    string Nombre { get; set; }
    string Email { get; set; }
}

public class Contacto : IReceptorMail
{
   public string Nombre { get; set; }
   public string CuentaBancaria { get; set; }
   public string Email { get; set; }
}
public interface IMailSender
{
   void Enviar(IReceptorMail contacto, string cuerpoMensaje);
}
```

En el Diagrama 9 se muestra como todas las clases dependen de la interfaz Usuario, cuando no todas hacen uso completo de ella.

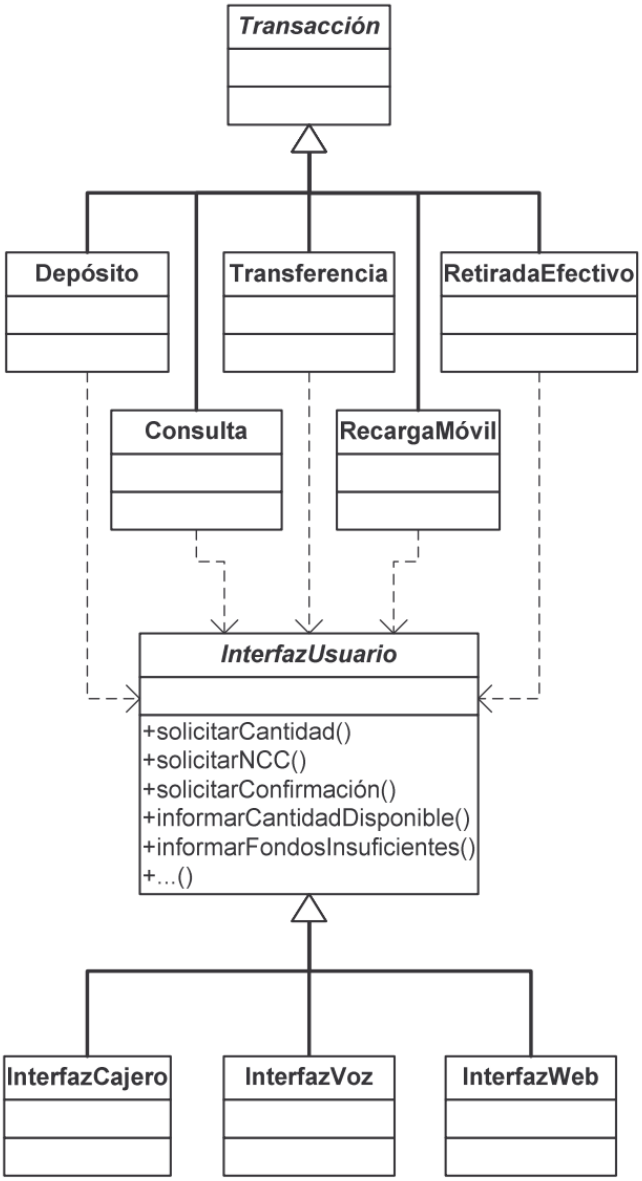

Diagrama 9. Diseño que no se ajusta al principio de segregación de la interfaz [17]

Para aplicar el principio se agregan más interfaces para cada una de las diferentes acciones, como se ilustra en el Diagrama 10, donde se creó una interfaz para cada una de las 5 acciones realizadas en una transacción.

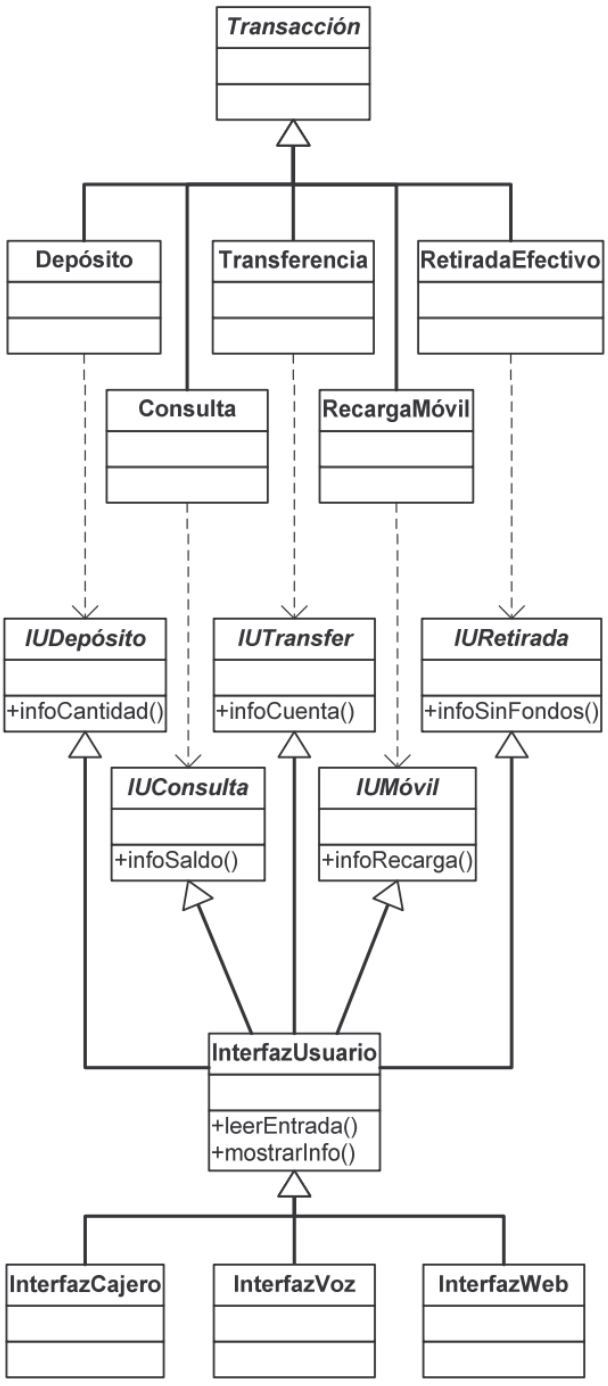

Diagrama 10. Diseño que si se ajusta al principio de segregación de la interfaz [17]

3.5 El Principio de Equivalencia de Versión y Reutilización

La esencia de la reutilización es la misma que de la versión. [3], [18]

Principio atribuido a Robert C. Martin. En el Principio de Equivalencia de Versión de Reutilización (REP-Release Reuse Equivalency Principle), un elemento reutilizable, sea un componente, una clase o un grupo de clases, no puede ser reutilizado a menos que esté vinculado a un sistema de lanzamiento de algún tipo. Ya que los paquetes son la unidad de lanzamiento, también son la unidad de reutilización. En el Diagrama 11 se muestra la relación de dependencia entre paquetes.

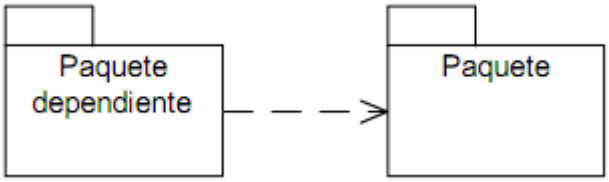

Diagrama 11. Dependencia entre paquetes

3.6 El Principio de Cierre Común

Las clases que cambian juntas, permanecen juntas. Los componentes que comparten funciones entre ellos o que dependen uno del otro deberían ser colocados juntos. [3], [18]

Principio atribuido a Robert C. Martin. En el Principio de Cierre Común (CCP-Common Closure Principle), un gran proyecto de desarrollo está subdividido en una extensa red de paquetes interrelacionados. Se pretende minimizar el número de paquetes que cambian en cada ciclo de lanzamiento del producto.

Las clases deben empaquetarse de manera que sean un todo coherente, es decir, cuando las clases se empaquetan como parte de un diseño, deben atender la misma área de funciones o comportamiento. Esto lleva a un control de cambio y a un manejo de versiones más efectivo.

3.7 El Principio de Reutilización Común

Las clases que no son reutilizadas juntas no deben estar agrupadas juntas. Si se reutiliza una clase de un paquete entonces se reutilizan todas. [3], [18]

Principio atribuido a Robert C. Martin. En el Principio de Reutilización Común (CRP-Common Reuse Principle), si esto sucede, los cambios a una clase sin relevancia igualmente forzarían un nuevo lanzamiento del paquete, provocando el esfuerzo necesario para su mejora y revalidación.

Cuando una o más clases cambian en un paquete, también cambia el número de versión del paquete. Todas las demás clases o paquetes que dependen de ese paquete deben actualizarse ahora a la versión más reciente del paquete y probarse para asegurar que la nueva versión funciona sin incidentes.

Si no hubo cohesión al agrupar las clases, es posible que cambie una clase que no tenga relación con las demás. Esto requerirá un proceso innecesario de integración y de prueba. Por esto, sólo deben incluirse en un mismo paquete las clases que se reutilizarán juntas.

3.8 El Principio de Dependencias Acíclicas

Las dependencias entre paquetes no deben formar ciclos. [3], [18]

Principio atribuido a Robert C. Martin. En el Principio de Dependencias Acíclicas (ADP-Acyclic Dependencies Principle), los ciclos se pueden romper de dos maneras. La primera involucra la creación de un nuevo paquete, mientras que la segunda hace uso del DIP (Principio de Inversión de la Dependencia) y el SIP. En el Diagrama 12, se ilustra un grupo de paquetes donde no existen ciclos.

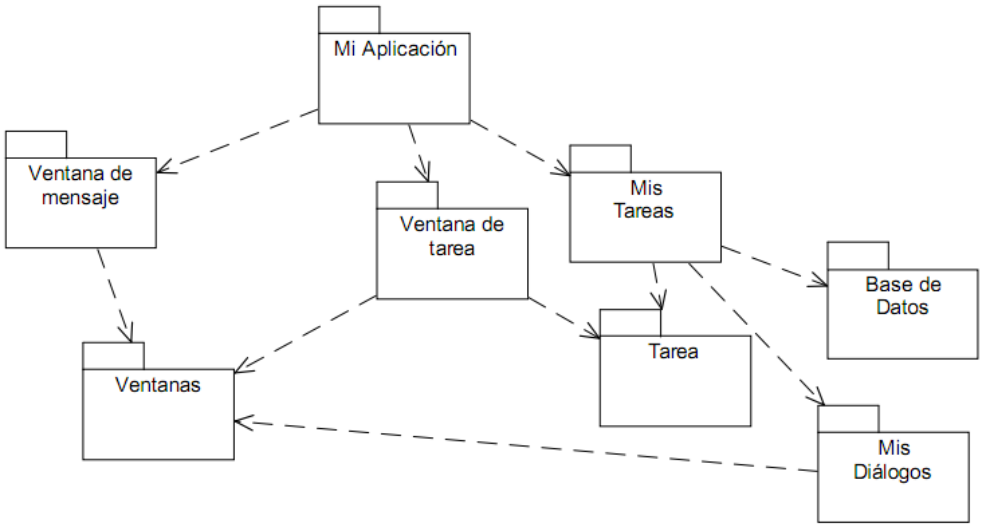

Diagrama 12. Paquetes sin ciclos [19]

En el Diagrama 13, se muestra un grupo de paquetes con ciclos de dependencia, donde existe una dependencia directa entre el paquete “Mi Aplicación” y el paquete “Mis Diálogos”, lo cual genera problemas si se realiza un cambio en cualquiera de los dos.

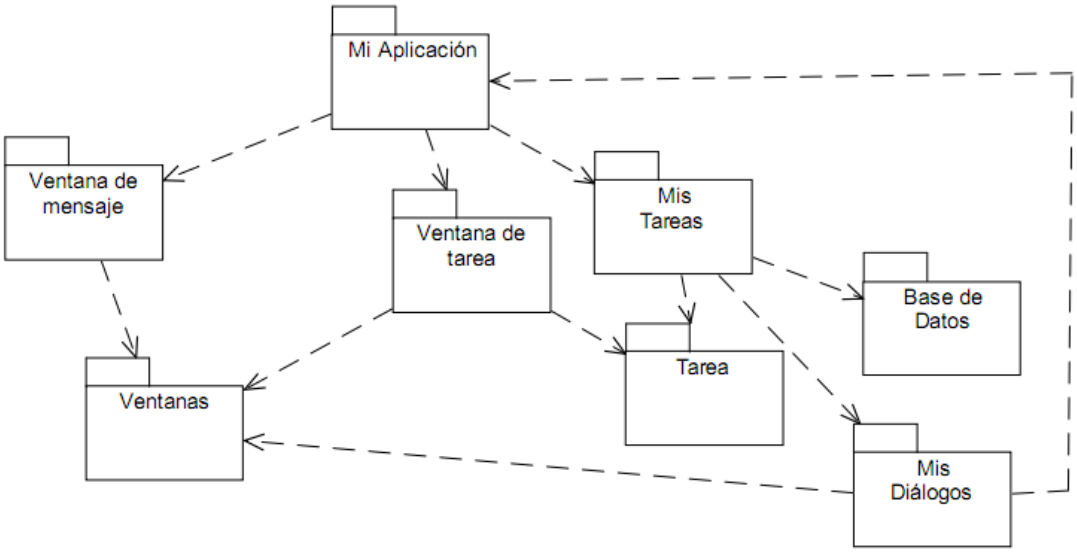

Diagrama 13. Paquetes con ciclos [19]

3.9 El Principio de Dependencias Estables

Depende de la dirección de la estabilidad. Un paquete debe depender sólo de los paquetes que son más estables que él. [3], [20]

Principio atribuido a Robert C. Martin. En el Principio de Dependencias Estables (SDP-Stable Dependencies Principle), se establece la dependencia en la dirección de estabilidad. La estabilidad está relacionada con la cantidad de trabajo necesario para realizar un cambio.

Medida de Estabilidad

Ca Aferente de acoplamiento. El número de clases fuera del paquete que depende de clases dentro del mismo.

Ce Eferente de acoplamiento. El número de clases fuera del paquete que dependen las clases dentro del mismo.

I Inestabilidad. Medida en el rango [0,1]

$$I = \frac{Ce}{Ca + Ce}$$

Dependa de paquetes cuya medida de inestabilidad sea menor a la suya.

3.10 El Principio de Abstracciones Estables

Paquetes estables deben ser paquetes abstractos. [3], [20]

Principio atribuido a Robert C. Martin. En el Principio de Abstracciones Estables (SAP-Stable Abstractions Principle), podemos visualizar la estructura de paquetes en nuestra aplicación como un conjunto de paquetes interconectados; los paquetes inestables arriba y los estables debajo. Luego, todas las dependencias apuntan hacia abajo.

Los paquetes que son estables al máximo deben ser abstractos al máximo. Los paquetes inestables deben ser concretos. La abstracción de un paquete debe ser proporcional a su estabilidad.

Medida de Abstracción

Nc Número de clases dentro del paquete.
Nc Número de clases abstractas dentro del paquete.
A Abstracción. Medida en el rango [0,1]

$$A = \frac{Na}{Nc}$$

En el Diagrama 14, se ilustra la relación directa que existe entre la abstracción y la inestabilidad, donde a mayor inestabilidad menor abstracción y viceversa a mayor abstracción menor inestabilidad.

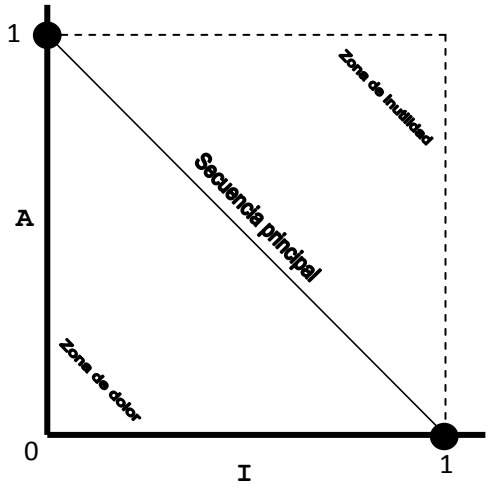

Diagrama 14. Relación abstracción contra inestabilidad

Medida de Alejamiento

D Distancia a la secuencia principal, lugar donde el paquete se proporciona entre la abstracción de sus dependencias entrantes y la implementación de sus dependencias salientes. Medida en el rango [0, ~0.707]

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

Estas medidas tallan la arquitectura orientada a objetos. Son imperfectas y depender únicamente de ellas como indicador de la firmeza de la arquitectura sería temerario. Sin embargo, pueden ser y han sido usadas como ayuda para medir la estructura de dependencias de una aplicación.

4 Análisis de los Principios

Para cada uno de los principios analizados se identificaron las mejores prácticas, las cuáles se resumen en la Tabla 3.

Principio	Mejores prácticas
Abierto Cerrado Un módulo debe ser abierto para extensión pero cerrado para modificación.	Usar composición y herencia para llevar a cabo modificaciones. Implementar una interfaz en todas las clases que son susceptibles de ser modificadas en un futuro. Usar polimorfismo.
De Sustitución de Liskov Subclases deben ser sustituibles por sus clases base.	Hacer uso de herencia. en las clases donde el comportamiento sea el mismo, ya que una clase no solo sustituye a otra por ser un tipo de ella sino también por comportarse como ella.
De Inversión de Dependencia Dependerá de abstracciones. No dependerá de implementaciones. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.	Hacer uso de abstracciones. Creación de constructores de clases de las que se depende.
De Segregación de la Interfaz Muchas interfaces específicas de los clientes son mejor que una sola interfaz de propósito general. Los clientes de una clase no deberían depender de interfaces que no utilizan.	Categorizar clientes por tipo y crear interfaces para cada tipo.
De Equivalencia de Versión y Reutilización La esencia de la reutilización es la misma que de la versión.	Agrupar las clases reutilizables en paquetes.
De Cierre Común Las clases que cambian juntas, permanecen juntas. Los componentes que comparten funciones entre ellos o que dependen uno del otro deberían ser colocados juntos.	Agrupar las clases que comparten funciones en paquetes.
De Reutilización Común Las clases que no son reutilizadas juntas no deben estar agrupadas juntas. Si se reutiliza una clase de un paquete entonces se reutilizan todas.	Solo agrupar en un paquete a las clases que se reutilizarán juntas.
De Dependencias Acíclicas Las dependencias entre paquetes no deben formar ciclos.	Cuando se generen las dependencias entre paquetes, nunca dirigir las dependencias al paquete origen.
De Dependencias Estables Depende de la dirección de la estabilidad. Un paquete debe depender sólo de los paquetes que son más estables que él.	Las dependencias entre paquetes en un diseño deben hacerse buscando la dirección de la estabilidad de los paquetes.
De Abstracciones Estables Paquetes estables deben ser paquetes abstractos.	Los paquetes que son estables al máximo deben ser abstractos al máximo. Los paquetes inestables deben ser concretos. La abstracción de un paquete debe ser proporcional a su estabilidad.

Tabla 3. Mejores prácticas relacionadas a los principios de diseño

Al momento de analizar cada uno de los principios se detecto la relación entre ellos, por lo que se infiere que cumpliéndose uno de los principios se cumple automáticamente otro, esta relación se resume en la Tabla 4.

Principio	SIGLAS	Relación con otros principios
1. Principio Abierto Cerrado	OCP	
2. Principio de Sustitución de Liskov	LSP	OCP
3. Principio de Inversión de Dependencia	DIP	
4. Principio de Segregación de la Interfaz	ICP	
5. Principio de Equivalencia de Versión y Reutilización	REP	
6. Principio de Cierre Común	CCP	CRP
7. Principio de Reutilización Común	CRP	CCP
8. Principio de Dependencias Acíclicas	ADP	
9. Principio de Dependencias Estables	SDP	
10. Principio de Abstracciones Estables	SAP	

Tabla 4. Relación entre los principios de diseño

De los principios propuestos por Robert Martin, se realizó un análisis, igual al hecho anteriormente para otros autores, en cuanto al ataque que tiene cada principio a los diferentes síntomas del diseño degradado, como se resume en la Tabla 5.

Síntoma	Principal Característica	Principio que lo ataca
Rigidez	Difícil de cambiar	• Principio de Sustitución de Liskov • Principio de Inversión de Dependencia • Principio de Dependencias Estables • Principio de Abstracciones Estables
Fragilidad	Falla en muchos lugares	• Principio Abierto Cerrado • Principio de Dependencias Acíclicas • Principio de Inversión de Dependencia
Inmovilidad	Falta de reutilización	• Principio de Sustitución de Liskov • Principio de Equivalencia de Versión y Reutilización • Principio de Cierre Común • Principio de Reutilización Común • Principio de Inversión de Dependencia
Viscosidad	No se preserva el diseño	• Principio de Inversión de Dependencia
Complejidad innecesaria	Infraestructura del diseño sin beneficios	• Principio de Segregación de la Interfaz
Repetición innecesaria	Código duplicado	• Principio de Cierre Común
Opacidad	Poco entendimiento, código sin estructurar	• Principio de Inversión de Dependencia

Tabla 5. Relación de principios con síntomas

5 Principios de Diseño aplicados a Árboles

5.1 Introducción a Árboles

En computación, un árbol es una estructura de datos que imita la forma de un árbol, la cual consiste en un conjunto de nodos conectados entre sí. Un nodo es el componente utilizado para construir un árbol, y puede tener cero o más nodos hijos conectados a él.

Habitualmente los árboles se dibujan desde el nodo raíz hacia abajo; exactamente lo opuesto a la manera en que crecen los árboles naturales.

El uso de las estructuras **árbol** ayudan a la organización de los datos, ya que permiten implementar un conjunto de algoritmos más rápidos que cuando se usan estructuras lineales de datos. Son ampliamente usados en sistemas de archivos, interfaces gráficos, sistemas de toma de decisiones, bases de datos, sitios Web, compiladores, sistemas de diagnostico medico y otros sistemas de cómputo.

Las relaciones que existen en un árbol son jerárquicas. La terminología utilizada para el manejo de árboles es la siguiente:

- **raíz:** Único nodo sin padre.
- **hijo**: También llamado descendiente. Se dice que X es hijo de Y, sí y solo sí el nodo X es apuntado por Y.
- **padre**: También llamado antecesor. Se dice que X es padre de Y sí y solo sí el nodo X apunta a Y.
- **hermano**: Dos nodos serán hermanos si son descendientes directos de un mismo nodo.
- **hoja**: También llamada terminal. Son aquellos nodos que no tienen ramificaciones (hijos).
- **rama:** Es un nodo que no es raíz ni hoja.
- **grado**: Es el número de descendientes directos de un determinado nodo.
- **grado de un árbol**: Es el máximo grado de todos los nodos del árbol.
- **nivel**: También llamado profundidad. Es el número de enlaces que deben ser recorridos para llegar a un determinado nodo. Por definición la raíz tiene el nivel de 1.
- **altura**: Es el máximo número de niveles de todos los nodos del árbol.
- **peso**: Es la cantidad de nodos del árbol sin contar la raíz.
- **longitud de camino**: Es el número de enlaces que deben ser recorridos para llegar desde la raíz al nodo X. Por definición la raíz tiene longitud de camino 1, y sus descendientes directos longitud de camino 2 y así sucesivamente.
- **generación**: Conjunto de nodos con el mismo nivel.

Los árboles se representan gráficamente, mediante:

- **Círculos y flechas,** como se ilustra en la Figura 3, donde A es la raíz, padre de B y C, y estos a su vez son padres de D y E, respectivamente. La representación mediante círculos y flechas es la más común, y la usada en este trabajo.

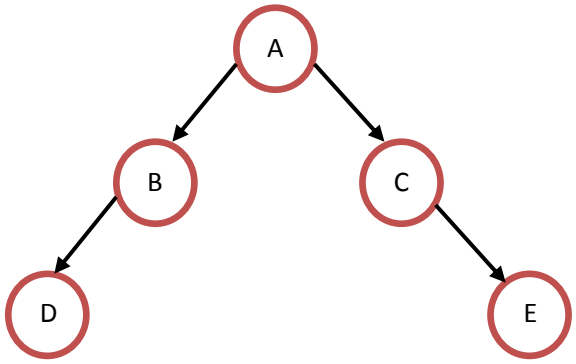

Figura 3. Representación de un árbol con círculos y flechas

- **Diagramas de Venn,** como se ilustra en la Figura 4. Donde D y E son hijos de B y C, respectivamente y A es la raíz del árbol y a su vez padre de B y C.

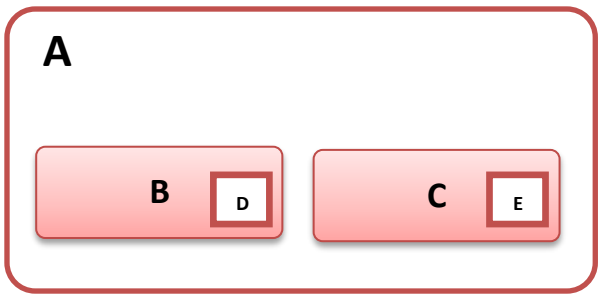

Figura 4. Representación de un árbol con un diagrama de Venn

- **Paréntesis anidados,** donde se inicia con la raíz y entre paréntesis se agregan sus hijos y así sucesivamente para el agregado de más nodos.

$$(\mathbf{A}(\mathbf{B}(\mathbf{D}),\mathbf{C}(\mathbf{E})))$$

Y se implementan en lenguajes de programación (Java) mediante:

- **Matriz,** como se ilustra en la Figura 5, la matriz correspondiente para el árbol de la Figura 3. Esta representación usa la propiedad de los árboles de que cada nodo tiene un único padre, por lo tanto la raíz A, tiene el valor de -1, B y C tienen solo un antecesor y D y E tienen 2.

A	-1	Raíz
B	1	
C	1	
D	2	
E	2	

Figura 5. Representación matricial de un árbol

- **Arreglos,** como se ilustra en la Figura 6, para cada elemento del árbol corresponde un índice dentro del arreglo.

1	2	3	4	5
A	B	C	D	E

Figura 6. Representación de un árbol mediante un arreglo

- **Lista de hijos,** como se ilustra en la Figura 7, donde A, que es la raíz, apunta a sus hijos B y D, de la misma manera B y C apuntan a D y E respectivamente, y los nodos hoja como D y E no apuntan a ningún nodo.

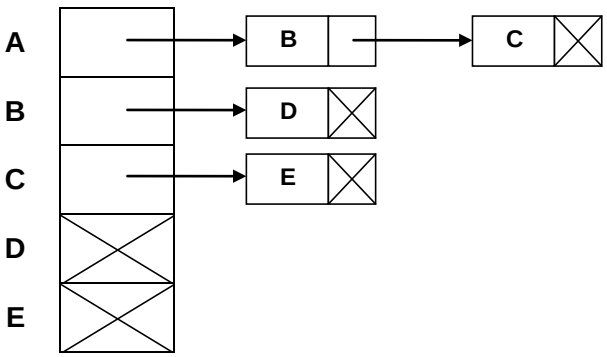

Figura 7. Representación de un árbol mediante listas de hijos

- **Listas enlazadas,** como se ilustra en la Figura 8, donde A es la raíz y su referencia izquierda apunta a su hijo izquierdo B, y su referencia derecha apunta a su hijo derecho C.

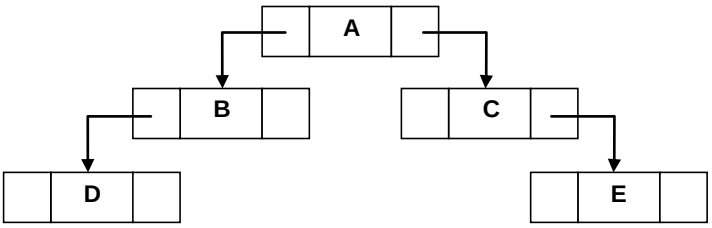

Figura 8. Representación de un árbol mediante listas enlazadas

Existen diferentes tipos de árboles, como se ilustra en la Figura 9, la principal clasificación se hace entre binarios y multicamino, donde para cada uno se desglosan otros más específicos.

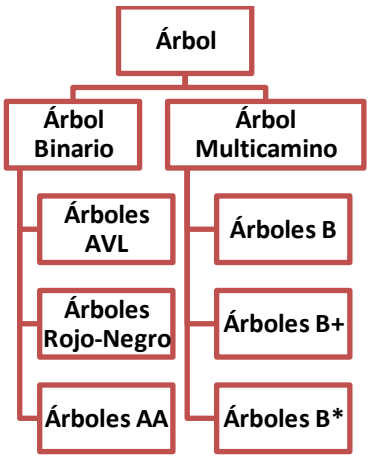

Figura 9. Clasificación de árboles

En este trabajo nos enfocaremos al diseño de **árboles binarios**, estos árboles contienen 2 enlaces en cada uno de sus nodos, uno de los cuales puede ser nulo. En la Figura 10, se ilustra un árbol binario representado con círculos y flechas.

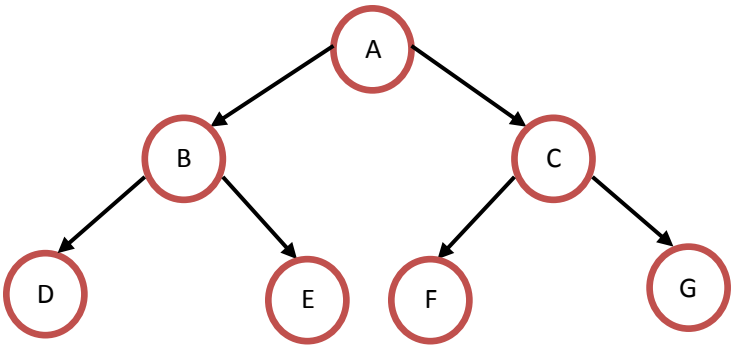

Figura 10. Representación con círculos y flechas de un árbol binario

5.2 Árboles Binarios de Búsqueda

Un árbol binario de búsqueda (ABB) es un tipo de un árbol binario, definido de la siguiente forma [21]:

Todo árbol vacío es un árbol binario de búsqueda.

Un árbol binario no vacío, de raíz R, es un árbol binario de búsqueda si:

- *En caso de tener subárbol izquierdo, la raíz R debe ser mayor que el valor máximo almacenado en el subárbol izquierdo, y que el subárbol izquierdo sea un árbol binario de búsqueda.*
- *En caso de tener subárbol derecho, la raíz R debe ser menor que el valor mínimo almacenado en el subárbol derecho, y que el subárbol derecho sea un árbol binario de búsqueda.*

Dentro de las operaciones realizadas en los árboles se encuentran:

- **Búsqueda:** El proceso de búsqueda inicia con el acceso a la raíz, donde se compara si el elemento buscado coincide, en este caso se concluye con éxito. En caso de que el elemento sea menor se pasa al subárbol izquierdo y si es mayor al subárbol derecho, y así sucesivamente hasta encontrar el elemento deseado. *Ver Apéndice C: Buscar.*

- **Inserción:** La inserción es muy parecida a la búsqueda. Si se tiene inicialmente un árbol vacío se crea un nuevo nodo y se inserta el elemento deseado. Si el elemento no se encuentra en el árbol, se comprueba si el elemento dado es menor que la raíz del árbol inicial con lo que se inserta en el subárbol izquierdo y si es mayor se inserta en el subárbol derecho y se comprueba que no estén ocupadas las dos hojas hijo del subárbol seleccionado, de ser así se repite el proceso hasta que la inserción se realice en un nodo hoja.

- **Eliminación:** La operación de eliminado se vuelve más compleja que las dos operaciones anteriores, dado que existen varios casos a tomar en cuenta:

    - **Eliminar un nodo sin hijos o nodo hoja:** se elimina el nodo y se establece a nulo el apuntador de su padre.

    - **Eliminar un nodo con un subárbol hijo:** se elimina el nodo y se asigna su subárbol hijo como subárbol de su padre.

    - **Eliminar un nodo con dos subárboles hijo**: aquí se reemplaza el valor del nodo por el de su predecesor o por el de su sucesor en inorden y posteriormente se borra el nodo. Su predecesor en inorden será el nodo más a la derecha de su subárbol izquierdo (mayor nodo del subárbol izquierdo), y su sucesor el nodo más a la izquierda de su subárbol derecho (menor nodo del subárbol derecho).

Otras de las operaciones dentro de un árbol binario de búsqueda son los recorridos, que es el orden en que se muestra o visita el árbol.

- Preorden:
  1. Se visita la raíz.
  2. Recorrer el subárbol izquierdo en preorden.
  3. Recorrer el subárbol derecho en preorden.

- Inorden:
  1. Recorrer el subárbol izquierdo en inorden.
  2. Se visita la raíz.
  3. Recorrer el subárbol derecho en inorden.

- Postorden:
  1. Recorrer el subárbol izquierdo en postorden.
  2. Recorrer el subárbol derecho en postorden.
  3. Se visita la raíz. *Ver Apéndice C: Recorridos.*

Para el caso del recorrido inorden se muestran los datos en orden ascendente.

Dentro de los árboles binarios de búsqueda más comunes encontramos los árboles AVL, árboles Rojo-Negro y árboles AA, como se ilustra en la Figura 11, estos 3 tipos de árboles son autobalanceables, esto quiere decir que la diferencia entre las alturas de sus subárboles no excede una unidad de diferencia, cada uno de estos árboles cuenta con un mecanismo propio para mantenerse equilibrado. En el caso del árbol AVL se cuenta con un factor de equilibrio en cada nodo, el árbol Rojo-Negro se basa en colores y el árbol AA usa el nivel de los nodos para mantenerse en equilibrio.

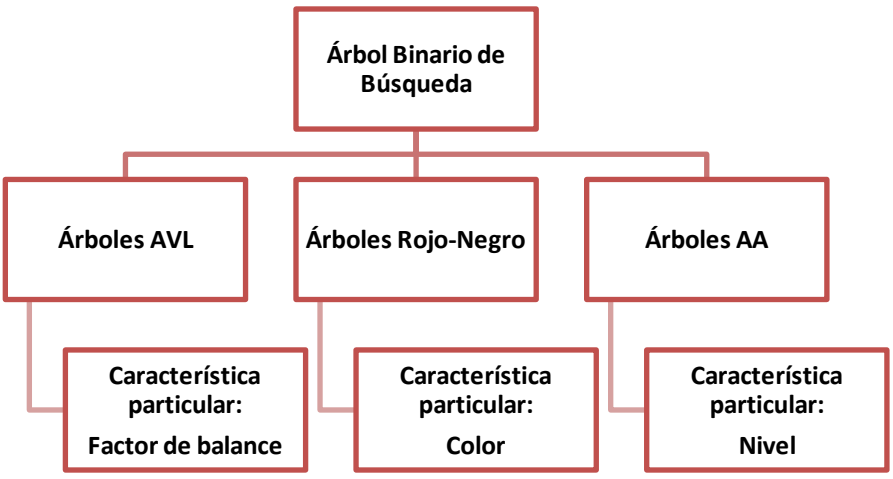

Figura 11. Clasificación de los árboles binarios de búsqueda

5.2.1 Árboles AVL

El árbol AVL toma su nombre de las iniciales de los apellidos de sus inventores, Adelson-Velskii y Landis. Lo dieron a conocer en la publicación de un artículo en 1962: " Un algoritmo para la organización de la información". [22]

Los árboles AVL son árboles equilibrados, esto quiere decir que para todos sus nodos la altura del subárbol izquierdo no difiere en más de una unidad la altura del subárbol derecho. Para controlar esta diferencia entre las alturas de los subárboles se cuenta con un factor de equilibrio para cada nodo.

Las operaciones de inserción y eliminación se realizan de manera especial para mantener el estado de equilibrio en el árbol, después de realizar cualquiera de estas dos operaciones se requiere verificar el equilibrio del árbol y de ser el caso, reequilibrarlo, para lo cual se hace uso de rotaciones en los nodos, llamadas rotaciones AVL, existen dos tipos de rotaciones: simple y doble, a su vez pueden ser a la derecha o a la izquierda. *Ver Apéndice D: Rotaciones.*

Rotación simple a la derecha. Dado un árbol o subárbol de raíz **R**, y de subárbol izquierdo **I** y subárbol derecho **D** con sus respectivos hijos, se forma un nuevo árbol cuya raíz es la raíz del subárbol izquierdo **I**, como nuevo subárbol izquierdo colocamos el hijo izquierdo **i**, del subárbol izquierdo **I**, y como hijo derecho creamos un nuevo árbol que tendrá como raíz, la raíz del árbol **R**, el hijo derecho **d** del subárbol izquierdo **I**, será ahora el hijo izquierdo y el hijo derecho **D** del árbol original permanece como hijo derecho, como se puede apreciar en la Figura 12.

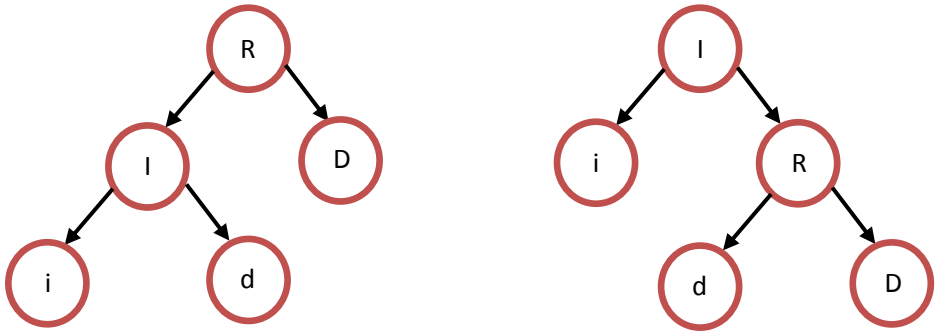

Figura 12. Rotación simple a la derecha.

Rotación simple a la izquierda. Dado un árbol o subárbol de raíz **R**, y de subárbol izquierdo **I** y subárbol derecho **D** con sus respectivos hijos, se forma un nuevo árbol cuya raíz es la raíz del subárbol derecho **D**, como nuevo subárbol derecho colocamos el hijo derecho **d**, del subárbol derecho **D**, y como hijo izquierdo creamos un nuevo árbol que tendrá como raíz, la raíz del árbol **R**, el hijo izquierdo **i** del subárbol derecho **D**, será ahora el hijo derecho y el hijo izquierdo **I** del árbol original permanece como hijo izquierdo, como se puede apreciar en la Figura 13.

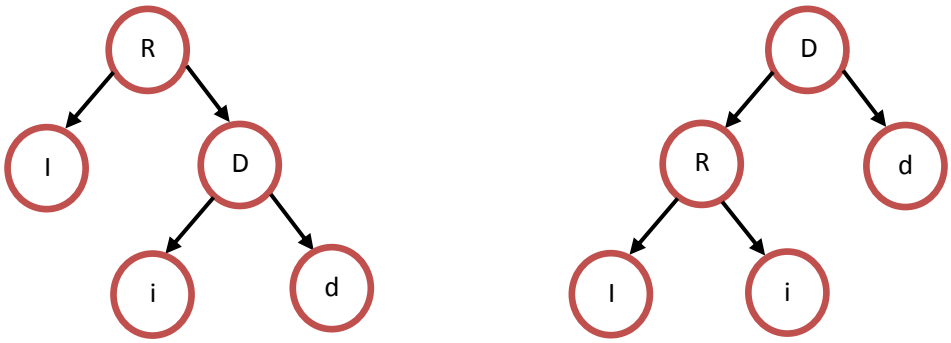

Figura 13. Rotación simple a la izquierda.

Rotación doble a la derecha. En esta rotación se realizan dos rotaciones simples, primero rotación simple izquierda y luego rotación simple derecha.

Rotación doble a la izquierda. En esta rotación se realizan dos rotaciones simples, primero rotación simple derecha y luego rotación simple izquierda.

Inserción. Al igual que en un árbol binario de búsqueda, la inserción en un árbol AVL se realiza comparando el elemento insertado con los nodos del árbol hasta encontrar la posición correspondiente dado su valor, posteriormente se tiene que comprobar el balance del árbol revisando el factor de balance del nodo, cuyo valor **absoluto** debe ser menor a 2, y se obtiene de la diferencia de alturas del subárbol izquierdo y derecho, de ser necesario se realiza el reajuste a través de las rotaciones que se presentaron anteriormente. *Ver Apéndice D: Insertar.*

Eliminación. El proceso de eliminación es igual al del árbol binario de búsqueda, pero al igual que en la inserción después de haber eliminado un nodo se tiene que comprobar el equilibrio del árbol revisando el factor de balance del nodo insertado, y aplicar las rotaciones necesarias para volver a equilibrarlo. *Ver Apéndice D: Eliminar.*

Para la representación gráfica de un árbol AVL se agrega el factor de balance en cada nodo debajo del valor, como se puede apreciar en la Figura 14.

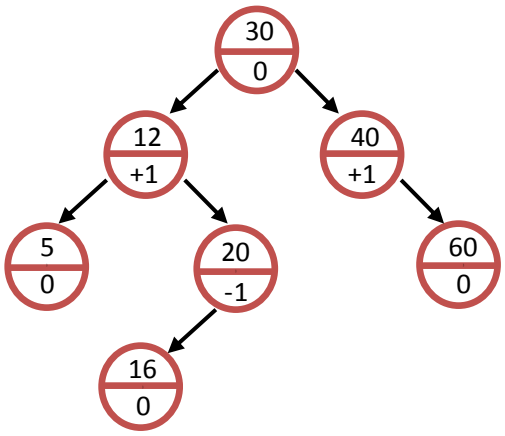

Figura 14. Ejemplo de árbol AVL

5.2.2 Árboles Rojo-Negro

Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo tiene un atributo de color cuyo valor es rojo o negro, la estructura original fue creada por Rudolf Bayer en 1972, que le dio el nombre de “árboles-B binarios simétricos”, pero tomó su nombre moderno en un trabajo de Leo J. Guibas y Robert Sedgewick realizado en 1978.

Además de los requisitos propios de los árboles binarios de búsqueda comunes, se deben satisfacer los siguientes criterios para tener un árbol rojo-negro válido [23]:

Todo nodo es o bien rojo o bien negro.

- *La raíz es negra.*
- *Todas las hojas son negras (las hojas son los hijos nulos).*
- *Los hijos de todo nodo rojo son negros (también llamada "Propiedad del rojo").*
- *Cada camino simple desde un nodo a una hoja descendiente contiene el mismo número de nodos negros, ya sea contando siempre los nodos negros nulos, o bien no contándolos nunca (el resultado es equivalente). También es llamada "Propiedad del camino", y al número de nodos negros de cada camino, que es constante para todos los caminos, se le denomina "Altura negra del árbol", y por tanto el camino no puede tener dos rojos seguidos.*
- *El camino más largo desde la raíz hasta una hoja no es más largo que 2 veces el camino más corto desde la raíz del árbol a una hoja en dicho árbol. El resultado es que dicho árbol está aproximadamente equilibrado.*

Para comprobar dichas propiedades, se revisa que ningún camino puede tener 2 nodos rojos seguidos.

El resultado de una inserción o eliminación de un nodo utilizando los algoritmos de un árbol binario de búsqueda normal podría violar las propiedades de un árbol rojo-negro. Restaurar las propiedades rojo-negro requiere un pequeño número de cambios de color y no más de 3 rotaciones (2 por inserción), según se requiera.

A diferencia del árbol AVL en las rotaciones se tiene que tomar en consideración los colores.

Rotación simple a la derecha. Dado un árbol o subárbol de raíz **R**, y de subárbol izquierdo **I** y subárbol derecho **D** con sus respectivos hijos, se forma un nuevo árbol cuya raíz es la raíz del subárbol izquierdo **I**, como nuevo subárbol izquierdo colocamos el hijo izquierdo **i**, del subárbol izquierdo **I**, y como hijo derecho creamos un nuevo árbol que tendrá como raíz, la raíz del árbol **R**, el hijo derecho **d** del subárbol izquierdo **I**, será ahora el hijo izquierdo y el hijo derecho **D** del árbol original permanece como hijo derecho. Posteriormente se realiza el cambio de colores según se requiera.

Rotación simple a la izquierda. Dado un árbol o subárbol de raíz **R**, y de subárbol izquierdo **I** y subárbol derecho **D** con sus respectivos hijos, se forma un nuevo árbol cuya raíz es la raíz del subárbol derecho **D**, como nuevo subárbol derecho colocamos

el hijo derecho **d**, del subárbol derecho **D**, y como hijo izquierdo creamos un nuevo árbol que tendrá como raíz, la raíz del árbol **R**, el hijo izquierdo **i** del subárbol derecho **D**, será ahora el hijo derecho y el hijo izquierdo **I** del árbol original permanece como hijo izquierdo. Posteriormente se realiza el cambio de colores según se requiera.

Rotación doble a la derecha. En esta rotación se realizan dos rotaciones simples, primero rotación simple izquierda y luego rotación simple derecha. Posteriormente se realiza el cambio de colores según se requiera.

Rotación doble a la izquierda. En esta rotación se realizan dos rotaciones simples, primero rotación simple derecha y luego rotación simple izquierda. Posteriormente se realiza el cambio de colores según se requiera.

Inserción. La inserción inicia añadiendo el nodo como lo haríamos en un árbol binario de búsqueda y pintándolo de rojo. Lo que sucede después depende del color de otros nodos cercanos. El término tío nodo será usado para referenciar al hermano del padre de un nodo, como en los árboles familiares humanos. *Ver Apéndice E: Insertar*

Eliminación. Para eliminar un nodo rojo seguimos el mismo procedimiento que para un árbol binario de búsqueda. El caso complejo es cuando el nodo que va a ser borrado y su hijo son negros, para este caso se deben revisar a detalle las propiedades del árbol. *Ver Apéndice E: Eliminar*

Para la representación gráfica de un árbol Rojo-Negro se agrega el color en cada nodo como se puede apreciar en la Figura 15.

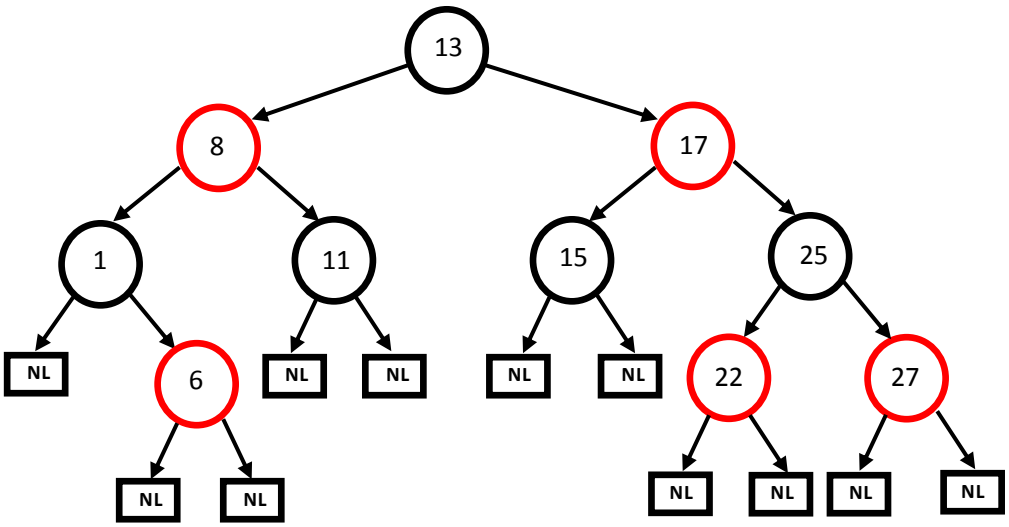

Figura 15. Ejemplo de árbol rojo-negro

5.2.3 Árboles AA

"Un árbol AA es un tipo de árbol binario de búsqueda auto-balanceable utilizado para almacenar y recuperar información ordenada de manera eficiente. Los árboles AA reciben el nombre de su inventor, Arne Andersson." [24]

Los árboles AA son una variación del árbol rojo-negro. A diferencia de los árboles rojo-negro, los nodos rojos en un árbol AA sólo pueden añadirse como un hijo derecho. En otras palabras, ningún nodo rojo puede ser un hijo izquierdo. Aunque el elemento **color no influye** al momento de reequilibrar el nodo.

Los árboles AA se implementan con la idea de un nivel en lugar de la de un color. Cada nodo tiene un campo nivel y se deben cumplir las siguientes condiciones para que el árbol sea válido [24]:

1. *El nivel de un nodo hoja es uno.*
2. *El nivel de un hijo izquierdo es estrictamente menor que el de su padre.*
3. *El nivel de un hijo derecho es menor o igual que el de su padre.*
4. *El nivel de un nieto derecho es estrictamente menor que el de su abuelo.*
5. *Cada nodo de nivel mayor que uno debe tener dos hijos.*

Sólo se necesitan dos operaciones para mantener el equilibrio en un árbol AA. Estas operaciones al igual que en los árboles anteriores son la rotación izquierda y derecha.

La rotación derecha se realiza cuando una inserción o una eliminación generan un enlace horizontal izquierdo. La rotación izquierda tiene lugar cuando una inserción o una eliminación crean dos enlaces horizontales derechos.

Inserción. La inserción inicia con la búsqueda normal en un árbol binario y su procedimiento de inserción. Después, a medida que se despliegan las llamadas, es fácil comprobar la validez del árbol y realizar las rotaciones que se necesiten. Si aparece un enlace horizontal izquierdo, se realiza una rotación derecha, y si aparecen dos enlaces horizontales derechos, se realiza una rotación izquierda, posiblemente incrementando el nivel del nuevo nodo raíz del subárbol correspondiente. *Ver Apéndice F: Insertar*

Eliminación. Para re-equilibrar un árbol existen diferentes aproximaciones. La que describió Andersson en su publicación original es la más simple, y consiste en que tras una eliminación, el primer paso para mantener la validez es reducir el nivel de todos los nodos cuyos hijos están dos niveles por debajo de ellos, o a los que les faltan hijos. Después, todo el nivel debe ser rotado a la derecha y posteriormente a la izquierda. *Ver Apéndice F: Eliminar*

Para la representación gráfica de un árbol AA se toma en cuenta el nivel de cada nodo, el cual se coloca al costado superior derecho del nodo, como se puede apreciar en la Figura 16.

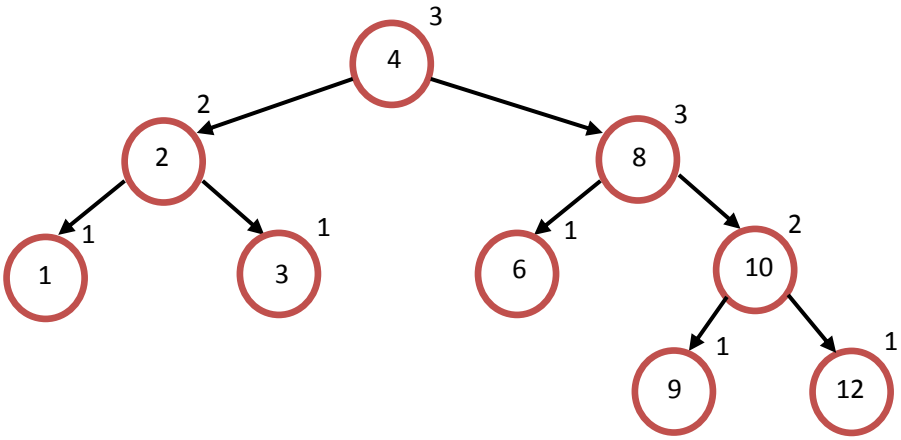

Figura 16. Ejemplo de árbol AA

5.2.4 Comparativa de los Árboles

Los tres tipos de árboles binarios de búsqueda explicados anteriormente cuentan con elementos que los distinguen, en el Árbol AVL, el reequilibrio depende del factor de balance en cada nodo, mientras que el Rojo-Negro toma en cuenta los colores en los nodos y el árbol AA depende del nivel del nodo. Por lo tanto al momento de implementarlos surgen ventajas y desventajas para el uso de cada uno, dependiendo de la aplicación que se desee hacer con el árbol, ya sea para buscar, devolver, borrar, insertar o balancear, existe una mejor opción resumida en la Tabla 6.

Características	Mejor opción árbol
Orden de complejidad	Todos cuenta con un orden de complejidad O log (n) Donde n es el número de elementos del árbol.
Tiempo de búsqueda	AVL
Tiempo de devolución	AVL
Tiempo de borrado	Rojo – Negro
Tiempo de inserción	Rojo – Negro
Balancear el árbol	AA

Tabla 6. Mejores opciones dada la característica a utilizar

5.3 Aplicación de Principios al Diseño de la Librería

Para este trabajo se creó una librería que contiene los 3 tipos de árboles binarios de búsqueda, mencionados anteriormente: AVL, Rojo-Negro y AA.

Las especificaciones para su representación de relación, acorde a las mejores prácticas definidas para cada principio están resumidas en la Tabla 7.

Concepto en UML	Expresa	Concepto en Diseño Orientado a Objetos	Representación Gráfica	Palabra reservada en Java
Generalización	Es un	Herencia		extends
Agregación	Tiene un	NA		NA
Realización	Usa	Interfaz		implements

Tabla 7. Representación de relaciones para aplicación de mejores prácticas

Al igual que las relaciones, se tienen que definir los elementos que conformaran cada una de las clases para el diseño de la librería, en el Diagrama 15 se muestra un ejemplo.

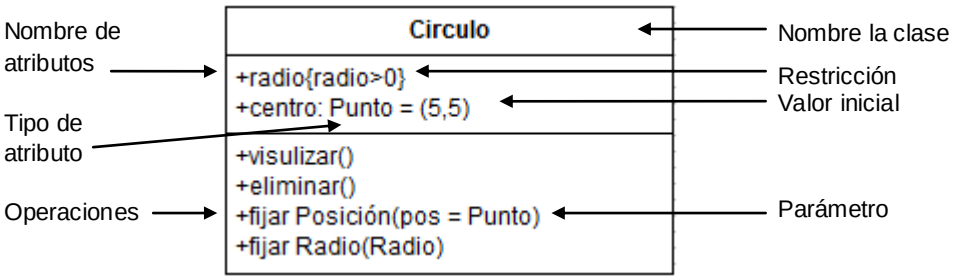

Diagrama 15. Ejemplificación de una clase en UML

Con los elementos especificados en el ejemplo anterior se definen los propios para cada clase que formará parte del diseño de la librería, los cuales se resumen en la Tabla 8.

Clase	Operaciones	Restricciones	Atributos

Nodo	Constructor Nodo	fb=0
		nivel=1
		color=Rojo
		Los nodos inicializados en null
			Nodo izquierdo
			Nodo derecho
			Nodo padre
			datos
			fb; // para AVL (factor de balanceo)
			color; // para RN
			nivel; // para AA (nivel del nodo)

Arboles Búsqueda	Constructor Arboles_Busqueda	NA	NA

	buscar mostrar re_inorde re_preorden re_postorden		
Árbol AVL	Constructor Arbol_AVL insertar altura decidir eliminar encontrarMaximo ajustar rotacion_Izquierda rotacion_Derecha	Los nodos inicializados en null	Nodo raíz Nodo insertado altura del subárbol izquierdo altura del subárbol derecho
Árbol Rojo-Negro	Constructor Rojo_Negro insertar cambiar_Color eliminar encontrarMaximo ajustar rotacion_Izquierda rotacion_Derecha	Los nodos inicializados en null	Nodo raíz Nodo insertado Nodo tío
Árbol AA	Constructor insertar eliminar encontrarMaximo ajustar rotacion_Izquierda rotacion_Derecha	Los nodos inicializados en null	Nodo raíz Nodo insertado
Interfaz Arbo	insertar eliminar rotación_Izquierda rotación_Derecha	NA	NA

Tabla 8. Definición de las clases

Establecidas las clases, se prosigue a definir qué métodos serán utilizados para herencia y cuales para interface.

Herencia

- La herencia evita redundancia de código y facilita su reutilización. Esta técnica permite el diseño de clases genéricas que se pueden especializar a clase más específicas. La herencia se aplica para los métodos que tienen **comportamiento** y propiedades idénticas. Para esto se obtienen los métodos presentados en la Tabla 9.

Métodos	Acción
buscar	Busca un elemento dado y responde si existe o no.
mostrar	Muestra los datos del árbol en cualquier recorrido preorden, inorden o postorden.
re_inorden	Recorre el árbol y muestra los datos en inorden.
re_preorden	Recorre el árbol y muestra los datos en preorden.
re_postorden	Recorre el árbol y muestra los datos en postorden.

Tabla 9. Métodos para herencia

Interfaz

- La interfaz declara un comportamiento común a todas las clases que implementan la interfaz. La interfaz se aplica para los métodos que se usan en diferentes clases pero que no tienen el mismo funcionamiento. Para esto se obtienen los métodos presentados en la Tabla 10.

Métodos	Acción
insertar	Dado un elemento recorre el árbol y busca la posición en que debe insertarse.
eliminar	Dado un elemento recorre el árbol y busca el elemento y lo elimina, posteriormente rebalancea el árbol.
rotar_izquierda	Rota los datos a la izquierda.
rotar_derecha	Rota los datos a la derecha.

Tabla 10. Métodos para interfaz

Una clase puede implementar más de un interfaz en Java, pero sólo puede extender una clase. Otra de las consideraciones a tomar en cuenta al momento de diseñar es el alcance de las variables y métodos según el uso que se les esté dando debemos tomar en cuenta la información mostrada en la Tabla 11.

Tipo de elemento	¿Accesible a clase de paquete?	¿Accesible a clase derivada?	¿Accesible a clase derivada de otro paquete?
public	si	si	si
protected	si	si	si
private	no	no	no
default	si	si	no

Tabla 11. Acceso a variables y métodos

Después de un exhaustivo análisis de cada uno de los árboles binarios y aplicando la representación de UML anteriormente mencionada, se genera un diagrama para la librería que contiene los tres tipos de árboles binarios de búsqueda, y que además hace uso de herencia e interfaz para la aplicación de los 10 Principios Fundamentales de Diseño, como se puede apreciar en el Diagrama 16.

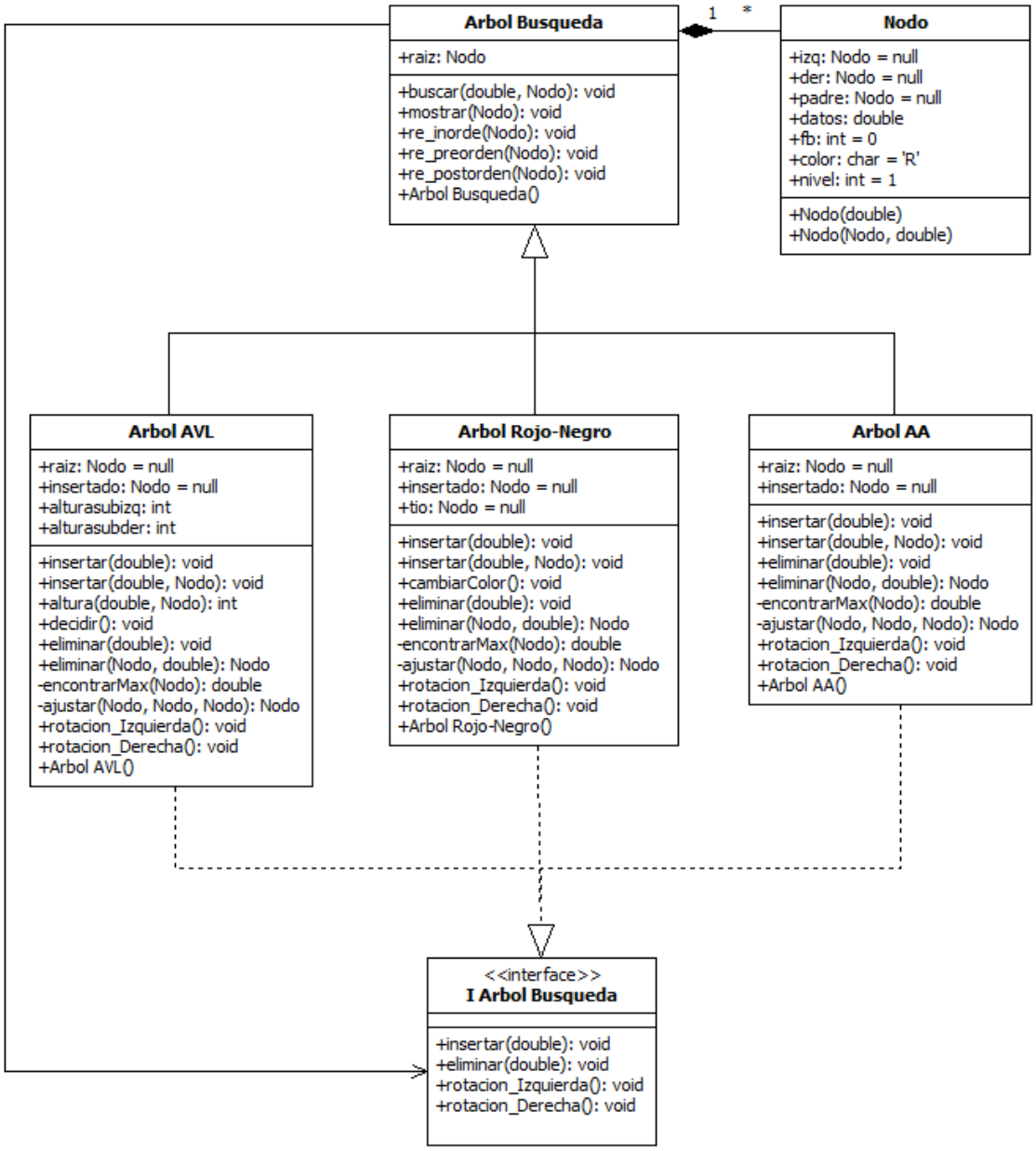

Diagrama 16. Diagrama de clases para la librería de árboles binarios de búsqueda.

La aplicación de cada uno de los principios de diseño en el diagrama anterior se explica en Tabla 12.

Principio de diseño	Implementación en el diseño
1. **Principio Abierto Cerrado**	Implementación de la interfaz **Árbol_Busqueda** la clase queda abierta para extensión pero se limita su modificación. Ya que los métodos insertar y eliminar son diferentes para cada tipo de árbol pero a su vez todos los implementan.
2. **Principio de Sustitución de Liskov**	Las clases **AVL**, **Rojo_Negro** y **AA** heredan todos los métodos de **Árbol_Búsqueda**, se pueden usar en su lugar.
3. **Principio de Inversión de Dependencia**	Creación de constructores en la clase **Nodo** ya que las clases que lo aplican no dependen directamente de esta clase, sino de sus abstracciones.
4. **Principio de Segregación de la Interfaz**	Para este caso solo se tiene un tipo de cliente **Arboles de Busqueda Autobalanceados**, por lo que solo se crea un tipo de interfaz para todos esos árboles, dicha interfaz solo cuenta con 2 métodos que son comunes a todas las clases que los implementan.
5. **Principio de Equivalencia de Versión y Reutilización**	Se agrupan todas las clases en un paquete dado que son reutilizables.
6. **Principio de Cierre Común**	Todas las clases son colocadas en un solo paquete, la clase base es **Árbol Búsqueda** y las demás clase Árbol dependen de ella.
7. **Principio de Reutilización de Común**	Mismas características que satisfacen el punto 5 y 6.
8. **Principio de Dependencias Acíclicas**	No existen dependencias acíclicas entre las clases creadas. La relación entre las clases es lineal.
9. **Principio de Dependencias Estables**	Dado que la dependencia es lineal se vuelve estable.
10. **Principio Abstracciones Estables**	En la aplicación de interfaz dentro del paquete las clases se vuelven abstractas y por lo tanto estables.

Tabla 12. Implementación de los principios

El diagrama diseñado fue codificado en Java para la creación de la librería que contiene a los 3 árboles binarios de búsqueda, y puede encontrarse el código correspondiente a dicha aplicación en los apéndices de este trabajo.

6 Conclusiones

Se lograron identificar buenas prácticas para cada uno de los principios, lo que hizo más sencilla su aplicación en la creación de la librería, aspectos sencillos como la identificación de los métodos y variables necesarias para cada tipo de árbol de búsqueda, ayudo a definir las relaciones de herencia e interfaz que se debían implementar.

El uso de los principios resulto muy acertado ya que se tenían varios métodos similares y con el análisis detallado, se lograron identificar y agrupar según su función para hacer de la reutilización un proceso más sencillo, al igual que las modificaciones futuras a la aplicación creada.

Los primeros 4 principios de diseño: Principio Abierto Cerrado, Principio de Sustitución de Liskov, Principio de Inversión de la Dependencia y Principio de Segregación de la Interfaz, son considerados los más importantes para este trabajo, puesto que se enfocan en la administración de dependencias entre los módulos creados. Los siguientes 6 principios: Principio de Equivalencia de Versión de Reutilización, Principio de Cierre Común, Principio de Reutilización Común, Principio de Dependencias Acíclicas, Principio de Dependencias Estables y Principio de Abstracciones Estables, se enfocan en la manipulación del paquete creado, y solo se verifica su aplicación.

Los 10 principios de diseño popularizados por Robert C. Martin fueron propuestos con anterioridad por otros autores como lo son, el Principio Abierto-Cerrado y el de Sustitución de Liskov, y cumplen con su objetivo, que es evitar un diseño degradado, aplicando abstracción, encapsulación, modularidad y jerarquía, que resultan ser conceptos básicos dentro del diseño de software, aunque muchas veces pasados por alto al momento de diseñar.

7 Trabajo Futuro

- Definir de manera más formal las mejores prácticas para cada uno de los principios, agregando descripción detallada de cómo deben de implementarse cada práctica.
- Definir principios para el diseño a más alto nivel, para su posible aplicación en diagramas de componentes, despliegue, estados, actividad, secuencia, colaboración, etc., ya que actualmente los principios con los que se cuentan solo se centran en el diseño de componentes.
- Dada la definición de principios para un diseño a más alto nivel, obtener las mejores prácticas formalizadas.

Referencias

[1] Diccionario de la lengua española. 2005. *Espasa-Calpe*. Diccionario. (Septiembre 2005), 1142.

[2] IEEE Std. 1471-2000. 2000. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE Software.* (September 2000), 3. [Online]. Available: http://www.win.tue.nl/~johanl/educ/2II45/Lit/software-architecture-std1471-2000.pdf

[3] Robert C. Martin. 2000. Design principles and design patterns. *Object Mentor*, Citeseer. [Online]. Available: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[4] Fernando Berzal. 2006. Apuntes de programación orientada a objetos en Java: fundamentos de programación y principios de diseño. ISBN: 84-611-1405-1. (OOP - Principios de diseño: Java), 6-10. [Internet]. Disponible en: http://courseware.ikor.org/java/pdf/AB-classes.pdf

[5] Georgina Tazón. 2005. Ingeniería de Software. *Universidad Rey Juan Carlos*, 4-6. [Internet]. Disponible en: http://www.escet.urjc.es/~gtazon/IS/ConceptosDiseno.pdf

[6] Ronald Correa. 2007. Desarrollo de software Orientado a Objetos. *Universidad de Nariño.* [Internet]. Disponible en: http://ronaldcorrea.nireblog.com/post/2007/10/09/desarrollo-de-software-orientado-a-objetos-parte-1

[7] Alan M. Davis. 1995. 201 principles of software development. *McGraw-Hill*. (March 1995).

[8] Anthony I. Wasserman. 1996. Toward a Discipline of Software Engineering. *IEEE Software*. 13, 6 (November 1996), 23-31.

[9] Roger Pressman. 2005. Ingeniería de Software: Un Enfoque Práctico. 6ta. Edición. *McGraw-Hill.* (July 2005).

[10] Bertrand Meyer. 1988. Object-Oriented Software Construction. *Prentice Hall.*

[11] Barbara Liskov. 1987. Keynote address - Data abstraction and hierarchy. *SIGPLAN Not.* 23, 5 (January 1987), 17-34.

[12] Fernando Berzal. 2006. Apuntes de programación orientada a objetos en Java: fundamentos de programación y principios de diseño. ISBN: 84-611-1405-1. (OOP - Principios de diseño: Java), 9-10. [Internet]. Disponible en: http://courseware.ikor.org/java/pdf/AB-classes.pdf

[13] Robert C. Martin. 2000. The Dependency Inversion Principle. *Object Mentor*, Citeseer. [Online]. Available: http://www.objectmentor.com/resources/articles/dip.pdf

[14] Fernando Berzal. 2006. Apuntes de programación orientada a objetos en Java: fundamentos de programación y principios de diseño. ISBN: 84-611-1405-1. (OOP - Principios de diseño: Java), 15. [Internet]. Disponible en: http://courseware.ikor.org/java/pdf/AB-classes.pdf

[15] Robert C. Martin. 2000. The Interface Segregation Principle. *Object Mentor*, Citeseer. [Online]. Available: http://www.objectmentor.com/resources/articles/isp.pdf

[16] Daniel Mazzini. 2011. Interface Segregation Principle o principio de segregación de interfaces. [Internet]. Disponible en: http://danielmazzini.blogspot.com/2011/04/interface-segregation-principle-o.html

[17] Fernando Berzal. 2006. Apuntes de programación orientada a objetos en Java: fundamentos de programación y principios de diseño. ISBN: 84-611-1405-1. (OOP - Principios de diseño: Java), 18-20. [Online]. Available: http://courseware.ikor.org/java/pdf/AC-interfaces.pdf

[18] Robert C. Martin. 2000. Granularity. *Object Mentor*, Citeseer. [Online]. Available: http://www.objectmentor.com/resources/articles/granularity.pdf

[19] Francisco J. García, et al. 1997. Análisis y Diseño Orientado al Objeto para Reutilización. *Universidad de Burgos.* 2.1.1. (Octubre 1997), 39-40. [Internet]. Disponible en: http://www.giro.infor.uva.es/oldsite/docpub/Door.pdf

[20] Robert C. Martin. 2000. Stability. *Object Mentor*, Citeseer. [Online]. Available:http://www.objectmentor.com/resources/articles/stability.pdf

[21] Colaboradores de Wikipedia. 2012. Árbol binario de búsqueda. *Wikipedia, La enciclopedia libre*. (Septiembre 2012). [Internet]. Disponible en: http://es.wikipedia.org/wiki/Árbol_binario_de_búsqueda

[22] Georgi M. AdelsonVelskii, Yevgeni M. Landis. 1962. An Algorithm for the Organization of Information. Article.

[23] Colaboradores de Wikipedia. 2012. Árbol rojo-negro. *Wikipedia, La enciclopedia libre*. (Agosto 2012). [Internet]. Disponible en: http://es.wikipedia.org/wiki/Árbol_rojo-negro

[24] Colaboradores de Wikipedia. 2012. Árbol AA [Internet]. Wikipedia, *La enciclopedia libre*. (Julio 2012). [Internet]. Disponible en: http://es.wikipedia.org/wiki/Árbol_AA

Apéndices

Apéndice A: Estructura de la Clase Nodo

```
public class Nodo
{
    double dato;
    Nodo izq=null;
    Nodo der=null;
    Nodo padre=null;
    char color='R';//n para negro y r para rojo
    int nivel=1;
    int fb=0;  //para árbol AVL
    Nodo(double data){
        dato=data;
        color='N';
}

public Nodo(Nodo pa, double data)
{
        dato=data;
        nivel=1;
        padre=pa;
        izq=der=null;
        fb=0;
        color='R';
    }
}
```

Apéndice B: Estructura de la Interfaz Arbol Búsqueda

```
public interface I_Arbol_Busqueda
{
    void insertar(double d)throws Exception;
    void eliminar(double num)throws Exception;
    void rotacion_Izquierda();
    void rotacion_Derecha();
}
```

Apéndice C: Operaciones Básicas de un Árbol de Búsqueda

Buscar

```
    public void buscar(double num)throws Exception
    {
        buscar(num, raiz);
    }

    private void buscar(double num, Nodo rz)
    {
        try
        {
        if((rz == null) | (rz.datos == num))
        {
            System.out.print("Se encontro el dato: "+rz.datos+" ");
            return;
        }
        else
        {
            if(num>rz.datos)
            {
                buscar(num, rz.der);//Busca en el nodo derecho
            }
            else
            {
```

```
                buscar(num, rz.izq);//Busca en el nodo izquierdo
            }
        }
        }
        catch(Exception e)
        {
            System.out.println("El elemento no se encuentra");
        }
    }//buscar
```

Mostrar

```
public static void mostrar()
    {
    //recorre en inorden el árbol
        if(raiz!=null)
        {
            re_inorden(raiz);        //Aqui se puede usar preorden, inorden o
postorden
        }
    } //mostrar
```

Recorridos

```
////////////////////////////////////////////////////////////////RECORRIDOS
    //Recorrido inorden
    public static void re_inorden(Nodo raiz)
    {
        if(raiz!=null)
        {
            re_inorden(raiz.izq);
            System.out.print("\n "+raiz.datos);
            System.out.print(" y su color "+raiz.color);
            re_inorden(raiz.der);
        }
    } //inorden

    //Recorrido preorden
    public static void re_preorden(Nodo raiz)
    {
        if(raiz!=null)
        {
            System.out.print("\n "+raiz.datos);

            re_preorden(raiz.izq);
            re_preorden(raiz.der);
        }
    } //preorden

    //Recorrido postorden
    public static void re_postorden(Nodo raiz)
    {
        if(raiz!=null)
        {
            re_postorden(raiz.izq);
            re_postorden(raiz.der);
            System.out.print("\n "+raiz.datos);
        }
    } //postorden
```

Apéndice D: Operaciones de un Árbol AVL

Insertar

```
public void insertar(double d)throws Exception
{
    double x=d;
    if(raiz==null)
    {
        raiz=new Nodo(d);
    }
    else
    {
        insertar(x, raiz);
    }
    decidir();
}

public void insertar(double dato, Nodo raiz) throws Exception
{
    int hactual=altura(dato, raiz);

    if (dato< raiz.dato)
    {
        if (hactual>alturasubizq)
        {
            alturasubizq=hactual;
        }
    }
    else{
        if(hactual>alturasubder)
        {
            alturasubder=hactual;
        }
    }
}//insertar

int altura(double datot, Nodo raizt)
{
    int hactual=0;

    boolean bandera=false;
    while((datot != raizt.dato)&&(bandera==false))
    {
        if(datot < raizt.dato)  {
            if(raizt.izq!=null)
                raizt= raizt.izq;
            else{
                raizt.izq= new Nodo(raizt, datot);
                bandera=true;
                insertado=raizt.izq;
            }
        }

        else if(datot > raizt.dato)
        {
            if(raizt.der!=null)
            {
                raizt= raizt.der;
            }
            else
            {
                raizt.der= new Nodo(raizt, datot);
                bandera=true;
                insertado=raizt.der;
            }
```

```
            }
            hactual++;
        }
        return hactual;
    }//altura

    void decidir()throws Exception
    {
        raiz.fb=alturasubder-alturasubizq;

        System.out.println("fb: "+raiz.fb);

        if(Math.abs(raiz.fb)>=2)
        {
            if(raiz.fb<=-2)  //Rotar a la derecha
            {
                System.out.println("En rotación derecha fb: es "+raiz.fb);
                rotacion_Derecha();
                System.out.println("El nuevo valor para fb:"+raiz.fb);
            }
            else if(raiz.fb>=2)  //Rotar a la izquierda
            {
                System.out.println("El rotacion izquierda de fb: es
"+raiz.fb);
                rotacion_Izquierda();
                System.out.println("El nuevo valor para fb:"+raiz.fb);
            }
        }
    }//decider
```

Eliminar

```
public void eliminar(double num)throws Exception
    {
        try
        {
            eliminar(raiz, num);
            if(Math.abs(raiz.fb)>=2)
            {
                decidir();
            }
        }
        catch (Exception e)
        {
            System.out.println("El elemento no existe en el árbol");
        }
    }

    public Nodo eliminar(Nodo rz, double num)throws Exception
    {
        if(rz.dato==num)// se desea borrar la raiz
        {
            if(rz.der == null && rz.izq == null){  // se desea borrar una hoja
                rz = null;
                return rz;
            }
            if(rz.der == null){  // se tiene un solo hijo(izquierdo)
                rz = rz.izq;
                return rz;    // el hijo ocupa el lugar del padre
            }

            if(rz.izq == null){  // se tiene un solo hijo(derecho)
                rz = rz.der;
                return rz;
            }
            // tiene dos hijos
            rz.dato = encontrarMax(rz.izq); // sera igual al nodo de mayor
valor en el sub-arbol izquierdo
```

```
            rz = ajustar(rz, rz.izq, rz);
            return rz;//el nodo igualado (de mayor valor) se debe eliminar
para que no quede repetido
        }
        //si no era el nodo que se queria borrar se aprovecha su ordenamiento
para buscarlo

        if(num>rz.dato)
        { //si el elemento buscado es mayor al del nodo actual
            rz.der = eliminar(rz.der, num);
            return rz; //lo busca recursivamente en los nodo mayores
        }
        // el elemento este en la izquierda

        rz.izq = eliminar(rz.izq, num);
        return rz;
    }//eliminar

    //funcion que encuentra el maximo
    private Double encontrarMax(Nodo rz)
    {
        if(rz.der == null) //si no hay un nodo con mayor valor retorna el
valor del nodo actual
            return rz.dato;
        // sigue buscando en los nodos de mayor valor
        return encontrarMax(rz.der);
    }//eliminar
    private Nodo ajustar(Nodo padre, Nodo hijo, Nodo ancestro)
    {
        if(hijo.der == null){
            if(padre.equals(ancestro)){
                padre.izq = hijo.izq;
                return padre;
            }
            padre.der = hijo.izq;
            return padre;
        }
        // sigue buscando en los nodos de mayor valor
        hijo = ajustar(hijo, hijo.der, ancestro);
        if(padre.equals(ancestro))
            padre.izq = hijo;
        else padre.der = hijo;
        return padre;
    }//ajustar
```

Rotaciones

```
//ROTACION IZQUIERDA ***********************************
    public void rotacion_Izquierda()
    {
        Nodo pa=insertado.padre;
        Nodo ab=pa.padre;

        if(ab.padre==null)
        {
            pa.izq=ab;
            ab.padre=pa;
            ab.der=null;
            raiz=pa;
        }
        else
        {
            Nodo bis=ab.padre;
            Nodo pai=ab.izq;
            Nodo pad=ab.der;
            Nodo abi=bis.izq;
            Nodo aux;
```

```
            Nodo insd;

            if(ab.der==pa)    //pa es el hijo derecho
            {
                if(ab.izq==null)
                {
                    ab.izq=bis;
                    bis.padre=ab;
                    abi.padre=bis;
                    bis.der=null;
                    raiz=ab;
                }
                else
                {
                    aux=pai;
                    ab.izq=bis;
                    bis.padre=ab;
                    bis.der=aux;
                    aux.padre=bis;
                    abi.padre=bis;
                    raiz=ab;
                }
            }
            else if(ab.izq==pa) //pa es el hijo izquierdo
            {
                System.out.println("Entro a Doble rotación izquierda");

                if(pa.der==null) //Si no tiene hijos no se requiere auxiliar
para guardar el valor
                {
                    pa.der=ab;
                    ab.padre=pa;
                    bis.der=pa;
                    pa.padre=bis;
                    ab.der=pad;
                    insd=pad;
                    ab.izq=null;
                    //ab.padre=null;
                    rotacion_Izquierda();
                }
                else     //cuando es el hijo derecho de un hijo izquierdo
                {
                    insd=insertado;
                    aux=insd;
                    pa.der=ab;
                    ab.padre=pa;
                    bis.der=pa;
                    pa.padre=bis;
                    ab.der=pad;
                    ab.izq=aux;
                    aux.padre=ab;
                    rotacion_Izquierda();
                }
            }
        }
    }//rotacion izquierda

//ROTACION DERECHA
*****************************************************************************
    public void rotacion_Derecha()
    {

        Nodo pa=insertado.padre;
        Nodo ab=pa.padre;

        if(ab.padre==null)
        {
            pa.der=ab;
```

```
            ab.padre=pa;
            ab.izq=null;
            raiz=pa;
        }
        else
        {
            Nodo bis=ab.padre;
            Nodo pai=ab.izq;
            Nodo pad=ab.der;
            Nodo abd=bis.der;
            Nodo aux;
            Nodo insd;


            if(ab.izq==pa)    //pa es el hijo derecho
            {
                if(ab.der==null)
                {
                    ab.der=bis;
                    bis.padre=ab;
                    abd.padre=bis;
                    bis.izq=null;
                    raiz=ab;
                }
                else
                {
                    aux=pad;
                    ab.der=bis;
                    bis.padre=ab;
                    bis.izq=aux;
                    aux.padre=bis;
                    abd.padre=bis;
                    raiz=ab;
                }
            }
            else if(ab.der==pa) //pa es el hijo izquierdo
            {
                System.out.println("Entro a Doble rotación derecha");

                if(pa.izq==null) //Si no tiene hijos no se requiere auxiliar
para guardar el valor
                {
                    pa.izq=ab;
                    ab.padre=pa;
                    bis.izq=pa;
                    pa.padre=bis;
                    ab.izq=pai;
                    insd=pai;
                    ab.der=null;
                    rotacion_Derecha();
                }
                else     //cuando es el hijo izquierdo de un hijo derecho
                {
                    insd=insertado;
                    aux=insd;
                    pa.izq=ab;
                    ab.padre=pa;
                    bis.izq=pa;
                    pa.padre=bis;
                    ab.izq=pai;
                    ab.der=aux;
                    aux.padre=ab;
                    rotacion_Derecha();
                }
            }
        }
    }//rotación derecha
```

Apéndice E: Operaciones de un Árbol Rojo-Negro

Insertar

```
public void insertar(double d)throws Exception
{
    double x=d;
    if(raiz==null)
    {
        raiz=new Nodo(d);
    }
    else
    {
        insertar(x, raiz);
    }
}

public void insertar(double dato, Nodo raiz) throws Exception
{
    if(dato<raiz.dato)
    {
        if(raiz.izq==null)
        {
            raiz.izq=new Nodo(raiz,dato);
            insertado=raiz.izq;
            if(raiz.padre!=null)
            {
                cambiaColor();
            }
        }
        else
        {
            raiz=raiz.izq;
            insertar(dato, raiz);
        }
    }
    else if(dato>raiz.dato)
    {
        if(raiz.der==null)
        {
            raiz.der = new Nodo(raiz,dato);
            insertado=raiz.der;
            if(raiz.padre!=null)
            {
                cambiaColor();
            }
        }
        else
        {
            raiz=raiz.der;
            insertar(dato, raiz);
        }
    }
}//insertar

void cambiaColor()
{  //recibe el nodo al que se le insertó, no el insertado
    if(insertado.padre.padre!=null)
    {
        if(insertado.padre.padre.der==insertado.padre)
        {
            tio=insertado.padre.padre.izq;
        }
        else
        {
            tio=insertado.padre.padre.der;
```

```
            }

            if(tio!=null)
            {
                if(tio.color=='R')
                {
                    insertado.color='N';
                    tio.color='N';
                    //System.out.println("Se    cambió    el    color    de    pa
"+insertado.padre.dato+" y de tio "+tio.dato);
                if(insertado.padre!=raiz)
                {
                        insertado.padre.color='R';
                }
                }
            }
            else
            {
                if(insertado.dato<insertado.padre.padre.dato)
                {
                    System.out.println("Rotación derecha.");
                    rotacion_Derecha();
                }
                else
                {
                    System.out.println("Rotación izquierda.");
                    rotacion_Izquierda();
                }
            }
        }
    }//cambiar Color
```

Eliminar

```
public void eliminar(double num)throws Exception
    {
        try
        {
            eliminar(raiz, num);
        }
        catch (Exception e)
        {
            System.out.println("El elemento no existe en el árbol");
        }
    }//eliminar

    public Nodo eliminar(Nodo rz, double num)throws Exception
    {
        if(rz.dato==num)// se desea borrar la raiz
        {
            if(rz.der == null && rz.izq == null)
            {  // se desea borrar una hoja
                if(rz.padre!=null)
                    if((rz.padre.izq==null)||(rz.padre.der==null))
                        rz.padre.color='R';
                rz = null;
                return rz;
            }
            if(rz.der == null){  // se tiene un solo hijo(izquierdo)
                rz = rz.izq;
                return rz;    // el hijo ocupa el lugar del padre
            }

            if(rz.izq == null){  // se tiene un solo hijo(derecho)
                rz = rz.der;
                return rz;
            }
            // tiene dos hijos
```

```
            rz.dato = encontrarMax(rz.izq); // sera igual al nodo de mayor
valor en el sub-arbol izquierdo
            rz = ajustar(rz, rz.izq, rz);
            return rz;//el nodo igualado (de mayor valor) se debe eliminar
para que no quede repetido
        }
        //si no era el nodo que se queria borrar se aprovecha su ordenamiento
para buscarlo

        if(num>rz.dato)
        { //si el elemento buscado es mayor al del nodo actual
            rz.der = eliminar(rz.der, num);
            return rz; //lo busca recursivamente en los nodo mayores
        }
        // el elemento este en la izquierda

        rz.izq = eliminar(rz.izq, num);
        return rz;
    }//eliminar

    //funcion que encuentra el maximo
    private Double encontrarMax(Nodo rz)
    {
        if(rz.der == null) //si no hay un nodo con mayor valor retorna el
valor del nodo actual
            return rz.dato;
        // sigue buscando en los nodos de mayor valor
        return encontrarMax(rz.der);
    }//eliminar
    private Nodo ajustar(Nodo padre, Nodo hijo, Nodo ancestro)
    {
        if(hijo.der == null){
            if(padre.equals(ancestro)){
                padre.izq = hijo.izq;
                return padre;
            }
            padre.der = hijo.izq;
            return padre;
        }
        // sigue buscando en los nodos de mayor valor
        hijo = ajustar(hijo, hijo.der, ancestro);
        if(padre.equals(ancestro))
            padre.izq = hijo;
        else padre.der = hijo;
        return padre;
    }//ajustar
```

Rotaciones

```
public void rotacion_Derecha()
{
        Nodo pa=insertado.padre;
        Nodo ab=pa.padre;
        if(insertado.dato<insertado.padre.dato)
        {
            if(ab.padre.izq==ab)
            {
                ab.padre.izq=pa;
            }
            else
            {
                ab.padre.der=pa;
                pa.der=ab;
                pa.der.izq=null;
                pa.padre= ab.padre;
                pa.der.padre=pa;
```

```
                pa.color='N';
                pa.der.color='R';
        }
        else
        {
            if(ab.padre.izq==ab)
            {
                ab.padre.izq=insertado;
            }
            else
            {
                ab.padre.der=insertado;
                insertado.der=ab;
                ab.padre=insertado;
                insertado.izq=pa;
                pa.der=null;
                ab.izq=null;
                pa.padre=insertado;
                insertado.color='N';
                insertado.der.color='R';
            insertado.izq.color='R';
        }
    }//rotacion derecha

    public void rotacion_Izquierda()
    {
        Nodo pa=insertado.padre;
        Nodo ab=pa.padre;
        if(insertado.dato>insertado.padre.dato)
        {
            if(ab.padre.der==ab)
            {
                ab.padre.der=pa;
            }
            else
            {
                ab.padre.izq=pa;
                pa.izq=ab;
                pa.padre= ab.padre;
                pa.izq.padre=pa;
                pa.izq.der=null;
                pa.color='N';
                pa.izq.color='R';
            }
        else
        {
            //Doble rotación
            if(ab.padre.der==pa)
            {
                ab.padre.der=insertado;
            }
            else
            {
                ab.padre.izq=insertado;
                insertado.izq=ab;
                ab.padre=insertado;
                insertado.der=pa;
                pa.izq=null;
                ab.der=null;
                pa.padre=insertado;
                insertado.color='N';
                insertado.der.color='R';
                insertado.izq.color='R';
            }
    }//rotacion izquierda
```

Apéndice F: Operaciones de un Árbol AA

Insertar

```
public void insertar(double d)throws Exception
        {
            double x=d;
            if(raiz==null)
            {
                raiz=new Nodo(d);
            }
            else
            {
                insertar(x, raiz);
                raiz.nivel++;
                Nodo pa=insertado.padre;
                Nodo ab=pa.padre;

                if(raiz.nivel>2)
                {
                    if(ab!=null){
                        if(ab.der==pa)
                        {
                            if(pa.der==insertado)//Existen 2 enlaces derechos
                            {
                                rotacion_Izquierda();
                                raiz.nivel--;
                            }
                        }
                        else if(pa.izq==insertado)//Se inserto un elemento
izquierdo
                        {
                            /*Nodo aux=pa;
                            ab.izq=insertado;
                            insertado.padre=ab;
                            insertado.der=aux;
                            aux.padre=insertado;*/
                            insertado.padre=ab;
                            pa.padre=insertado;
                            insertado.der=pa;
                            ab.izq=insertado;
                            pa.izq=null;
                        }
                    }
                }
            }
        }

        public void insertar(double dato, Nodo raiz) throws Exception
        {
            if(dato<raiz.dato){
                if(raiz.izq==null){
                    raiz.izq=new Nodo(raiz,dato);
                    insertado=raiz.izq;
                }
                else{
                    raiz=raiz.izq;
                    insertar(dato, raiz);
                }
            }
            else if(dato>raiz.dato){
                if(raiz.der==null){
                    raiz.der = new Nodo(raiz,dato);
                    insertado=raiz.der;
                }
                else{
                    raiz=raiz.der;
```

```
                    insertar(dato, raiz);
                }
            }
        }//insertar
```

Eliminar

```
public void eliminar(double num)throws Exception
        {
            try
            {
                eliminar(raiz, num);
                raiz.nivel--;
            }
            catch (Exception e)
            {
                System.out.println("El elemento no existe en el árbol");
            }
        }

        public Nodo eliminar(Nodo rz, double num)throws Exception
        {
            if(rz.dato==num)// se desea borrar la raiz
            {
                if(rz.der == null && rz.izq == null){  // se desea borrar una
hoja
                    rz = null;
                    return rz;
                }
                if(rz.der == null){  // se tiene un solo hijo(izquierdo)
                    rz = rz.izq;
                    return rz;    // el hijo ocupa el lugar del padre
                }

                if(rz.izq == null){  // se tiene un solo hijo(derecho)
                    rz = rz.der;
                    return rz;
                }
                // tiene dos hijos
                rz.dato = encontrarMax(rz.izq); // sera igual al nodo de mayor
valor en el sub-arbol izquierdo
                rz = ajustar(rz, rz.izq, rz);
                return rz;//el nodo igualado (de mayor valor) se debe eliminar
para que no quede repetido
            }
            //si no era el nodo que se queria borrar se aprovecha su
ordenamiento para buscarlo

            if(num>rz.dato)
            { //si el elemento buscado es mayor al del nodo actual
                rz.der = eliminar(rz.der, num);
                return rz; //lo busca recursivamente en los nodo mayores
            }
            // el elemento este en la izquierda

            rz.izq = eliminar(rz.izq, num);
            return rz;
        }//eliminar

        //funcion que encuentra el maximo
        private Double encontrarMax(Nodo rz)
        {
            if(rz.der == null) //si no hay un nodo con mayor valor retorna el
valor del nodo actual
                return rz.dato;
            // sigue buscando en los nodos de mayor valor
            return encontrarMax(rz.der);
        }//eliminar
```

```
        private Nodo ajustar(Nodo padre, Nodo hijo, Nodo ancestro)
        {
            if(hijo.der == null){
                if(padre.equals(ancestro)){
                    padre.izq = hijo.izq;
                    return padre;
                }
                padre.der = hijo.izq;
                return padre;
            }
            // sigue buscando en los nodos de mayor valor
            hijo = ajustar(hijo, hijo.der, ancestro);
            if(padre.equals(ancestro))
                padre.izq = hijo;
            else padre.der = hijo;
            return padre;
        }//ajustar
```

Rotaciones

```
public void rotacion_Izquierda()
        {
            Nodo pa=insertado.padre;
            Nodo ab=pa.padre;

            if(ab.padre==null)
            {
                pa.izq=ab;
                ab.padre=pa;
                ab.der=null;
                raiz=pa;
            }
            else
            {
                Nodo bis=ab.padre;
                Nodo pai=ab.izq;
                Nodo pad=ab.der;
                Nodo abi=bis.izq;
                Nodo aux;
                Nodo insd;

                if(ab.der==pa)    //pa es el hijo derecho
                {
                    if(ab.izq==null)
                    {
                        ab.izq=bis;
                        bis.padre=ab;
                        abi.padre=bis;
                        bis.der=null;
                        raiz=ab;
                    }
                    else
                    {
                        aux=pai;
                        ab.izq=bis;
                        bis.padre=ab;
                        bis.der=aux;
                        aux.padre=bis;
                        abi.padre=bis;
                        raiz=ab;
                    }
                }
                else if(ab.izq==pa) //pa es el hijo izquierdo
                {
                    System.out.println("Entro a Doble rotación izquierda");
```

```
                    if(pa.der==null) //Si no tiene hijos no se requiere
auxiliar para guardar el valor
                    {
                        pa.der=ab;
                        ab.padre=pa;
                        bis.der=pa;
                        pa.padre=bis;
                        ab.der=pad;
                        insd=pad;
                        ab.izq=null;
                        //ab.padre=null;
                        rotacion_Izquierda();
                    }
                    else     //cuando es el hijo derecho de un hijo izquierdo
                    {
                        insd=insertado;
                        aux=insd;
                        pa.der=ab;
                        ab.padre=pa;
                        bis.der=pa;
                        pa.padre=bis;
                        ab.der=pad;
                        ab.izq=aux;
                        aux.padre=ab;
                        rotacion_Izquierda();
                    }
                }
            }
        }//rotacion izquierda

        public void rotacion_Derecha()
        {
            Nodo pa=insertado.padre;
            Nodo ab=pa.padre;
            Nodo aux=pa;
            ab.izq=insertado;
            insertado.padre=ab;
            insertado.der=aux;
            aux.padre=insertado;
        }
```

Apéndice G: Aplicación de la Librería

El ambiente de desarrollo para la librería fue IntelliJ IDEA Community Edition 11.1.2, cualquier posible cambio futuro se puede realizar mediante esta herramienta o cualquier otro editor de Java.

La aplicación realizada recibe de entrada datos dobles, hasta que se le agregue un número cero. Cuenta con las opciones de mostrar, buscar, insertar y eliminar un elemento. Dentro del código quedaron comentadas las opciones para hacer uso del árbol AVL, Rojo-Negro o AA, según se desee.

```
avl.insertar(avl.raiz);
//rn.insertar(rn.raiz);
//aa.insertar(aa.raiz);
```

En este caso se está implementando el método insertar de la clase AVL.

```
public class Aplicado
{
    public static void main(String args[])throws Exception
    {
        double d;
        BufferedReader entrada=new BufferedReader(new
InputStreamReader(System.in));
        AVL avl=new AVL();
        Rojo_Negro rn=new Rojo_Negro();
        AA aa=new AA();
        //Se crea el árbol con los datos
        do
        {
            System.out.println("Dato de entrada");
            System.out.flush();
            d=Double.parseDouble(entrada.readLine());

            if(d!=0)
            {
                avl.insertar(d);
                //rn.insertar(d);
                //aa.insertar(d);
            }
        }
        while(d!=0);

        do{
            System.out.println("\n\n");
            System.out.println("1. Mostrar el árbol ");
            System.out.println("2. Buscar un elemento");
            System.out.println("3. Insertar un elemento");
            System.out.println("4. Eliminar un elemento");
            System.out.println("5. Salir \n");
            System.out.flush();
            do{
                d=Double.parseDouble(entrada.readLine());
            }
            while(d<1||d>5);
            if(d==1)
            {
                System.out.println("Registros ordenados");
                avl.mostrar(avl.raiz);
                //rn.mostrar(rn.raiz);
                //aa.mostrar(aa.raiz);
            }
            else if(d==2)
            {
                double buscado;
                System.out.println("Número buscado");
                System.out.flush();
                buscado=Double.parseDouble(entrada.readLine());
                try
                {
                    avl.buscar(buscado, avl.raiz);
                    //rn.buscar(buscado,rn.raiz);
                    //aa.buscar(buscado,aa.raiz);
                }
                catch (Exception e)
                {
                    System.err.println(e);
                }
            }

            else if(d==3)
            {
                double nuevo;
```

```
                System.out.println("Número a insertar");
                System.out.flush();
                nuevo=Double.parseDouble(entrada.readLine());
                try
                {
                    avl.insertar(nuevo);
                    //rn.insertar(nuevo);
                    //aa.insertar(nuevo);
                }
                catch (Exception e)
                {
                    System.err.println(e);
                }
            }

            else if(d==4)
            {
                double dat;
                System.out.println("Número a eliminar");
                System.out.flush();
                dat=Double.parseDouble(entrada.readLine());
                try
                {
                    avl.eliminar(dat);
                    //rn.eliminar(dat);
                    //aa.eliminar(dat);
                }
                catch (Exception e)
                {
                    System.err.println(e);
                }
            }
        }
        while (d!=5);
    }
}//Aplicado
```

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS, A.C.

BIBLIOTECA

AUTORIZACION
PUBLICACION EN FORMATO ELECTRONICO DE TESIS

El que suscribe

Autor(s) de la tesis: Elizabeth Acuña Cháirez

Título de la tesis: Reporte Técnico "Principios de Diseño aplicados a Árboles Binarios de Búsqueda".

Institución y Lugar: Centro de Investigación en Matemáticas, A. C., unidad Zacatecas.

Grado Académico: Licenciatura ( ) Maestría (x) Doctorado ( ) Otro ( )

Año de presentación: 2012

Área de Especialidad: Ingeniería de Software

Director(es) de Tesis: Dr. Jorge Roberto Manjarrez Sánchez

Correo electrónico: elaicz@cimat.mx

Domicilio: zaragoza # 2, colonia Centro, Tlaltenango, Zacatecas.

Palabra(s) Clave(s): Diseño de software, componentes, modularidad, herencia, interfaz, diagramas UML, árboles binarios.

Por medio del presente documento autorizo en forma gratuita a que la Tesis arriba citada sea divulgada y reproducida para publicarla mediante almacenamiento electrónico que permita acceso al público a leerla y conocerla visualmente, así como a comunicarla públicamente en la Página WEB del CIMAT.

La vigencia de la presente autorización es por un periodo de 3 años a partir de la firma de presente instrumento, quedando en el entendido de que dicho plazo podrá prorrogar automáticamente por periodos iguales, si durante dicho tiempo no se revoca la autorización por escrito con acuse de recibo de parte de alguna autoridad del CIMAT

La única contraprestación que condiciona la presente autorización es la del reconocimiento del nombre del autor en la publicación que se haga de la misma.

Atentamente

_____

Nombre y firma del tesista

CALLE JALISCO S/N MINERAL DE VALENCIANA APDO. POSTAL 402
C.P. 36240 GUANAJUATO, GTO., MÉXICO
TELÉFONOS (473) 732 7155, (473) 735 0800 EXT. 49609 FAX. (473) 732 5749
E-mail: biblioteca@cimat.mx