Informe: Implementación de Agregación en Java

Emiliano Mendoza Montoya

Ficha 2848939

Análisis y Desarrollo de Software

Resumen

Este informe presenta un análisis exhaustivo de la implementación de agregación en Java a través del proyecto "Comprobantes". Se examina en profundidad la estructura de clases, las relaciones de agregación, la funcionalidad del código generado y las implicaciones prácticas de este diseño. El proyecto demuestra cómo la agregación facilita la composición de objetos complejos, la reutilización de código y la modelación de relaciones "tiene un" en programación orientada a objetos.

Introducción

La agregación es un concepto fundamental en la programación orientada a objetos que permite crear relaciones entre objetos donde un objeto contiene o está compuesto por otros objetos. Esta relación es menos fuerte que la herencia, permitiendo una mayor flexibilidad en el diseño de sistemas. El proyecto "Comprobantes" sirve como un caso de estudio práctico para ilustrar la aplicación de la agregación en Java, modelando las relaciones entre clases como `Factura`, `Cliente`, `Producto`, y `Fecha`.

Objetivos del Estudio

- Analizar la implementación de agregación en un contexto práctico de facturación.
- Evaluar la eficacia de la agregación en la composición de objetos complejos.
- Identificar las ventajas y posibles desafíos de utilizar agregación en el diseño de software.

Metodología

El análisis se llevó a cabo siguiendo estos pasos:

1. Examen detallado del diagrama de clases UML proporcionado.
2. Análisis del código fuente Java implementado.
3. Implementación y prueba del proyecto en un entorno de desarrollo Java.
4. Evaluación de la estructura de clases y sus interrelaciones.
5. Revisión de la funcionalidad demostrada en la clase principal `AppAgregacion`.

Análisis

Estructura de Clases

El proyecto "Comprobantes" implementa una estructura de clases que refleja un sistema de facturación:

1. **Clase** `Fecha`
   - Atributos: `dia`, `mes`, `anio` (todos `int`)
   - Métodos: Constructor, getters y setters
2. **Clase** `Comprobante`
   - Atributos: `tipo` (char), `numero` (int), `fecha` (Fecha)
   - Métodos: Constructor, getters y setters
3. **Clase** `Producto`
   - Atributos: `codigo` (int), `descripcion` (String), `precio` (float)
   - Métodos: Constructor, getters y setters
4. **Clase** `Cliente`
   - Atributos: `codigo` (int), `razonSocial` (String)
   - Métodos: Constructor, getters y setters
5. **Clase** `Factura` (extiende `Comprobante`)
   - Atributos adicionales: `total` (float), `mCliente` (Cliente), `mProducto` (ArrayList<Producto>)
   - Métodos: Constructor, getters, setters, `agregarProducto()`, `mostrarProducto()`, `mostrar()`

Implementación de la Agregación

La agregación se implementa principalmente en la clase `Factura`:

1. Agregación de `Cliente`: `private Cliente mCliente;`
2. Agregación de `Producto`: `private ArrayList<Producto> mProducto;`
3. Herencia de `Comprobante`, que a su vez agrega `Fecha`

Esta estructura permite:

- Composición flexible de objetos: Una factura contiene un cliente y múltiples productos.
- Reutilización de código: Las clases `Cliente`, `Producto`, y `Fecha` pueden ser utilizadas en otros contextos.
- Modelado de relaciones "tiene un": Una factura tiene un cliente, tiene productos, y tiene una fecha.

Análisis del Código

Clase `Fecha`

```
java
Copy
public class Fecha {
    private int dia;
    private int mes;
    private int anio;

    public Fecha(int dia, int mes, int anio) {
        setDia(dia);
        setMes(mes);
        setAnio(anio);
    }

// Getters y setters
}
```

La clase `Fecha` es una clase simple que encapsula los componentes de una fecha. Utiliza métodos setter en el constructor, lo que permite validación de datos al momento de la creación del objeto.

Clase `Producto`

```
java
Copy
public class Producto {
    private int codigo;
    private String descripcion;
    private float precio;

    public Producto(int codigo, String descripcion, float precio) {
        setCodigo(codigo);
        setDescripcion(descripcion);
        setPrecio(precio);
    }

// Getters y setters
}
```

`Producto` representa un ítem individual en la factura. La estructura permite una fácil extensión para incluir más atributos o métodos relacionados con el producto en el futuro.

Clase `Cliente`

```
java
Copy
public class Cliente {
    private int codigo;
    private String razonSocial;

    public Cliente(int codigo, String razonSocial) {
        setCodigo(codigo);
        setRazonSocial(razonSocial);
    }

// Getters y setters
}
```

La clase `Cliente` es sencilla pero efectiva para el propósito del sistema. Podría extenderse fácilmente para incluir más información del cliente si fuera necesario.

Clase `Comprobante`

```
java
Copy
public class Comprobante {
    private char tipo;
    private int numero;
    private Fecha fecha;

    public Comprobante(char tipo, int numero, Fecha fecha) {
        setTipo(tipo);
        setNumero(numero);
        setFecha(fecha);
    }

// Getters y setters
}
```

`Comprobante` es una clase base que podría utilizarse para diferentes tipos de comprobantes. Demuestra agregación al incluir un objeto `Fecha`.

Clase `Factura`

```
java
Copy
public class Factura extends Comprobante {
    private float total;
```

```
    private Cliente mCliente;
    private ArrayList<Producto> mProducto;

    public Factura(char tipo, int numero, Fecha fecha, Cliente client
e) {
        super(tipo, numero, fecha);
        setCliente(cliente);
        mProducto = new ArrayList<>();
    }

    public void agregarProducto(Producto p) {
        mProducto.add(p);
        setTotal(getTotal() + p.getPrecio());
    }

    public void mostrarProducto() {
        Iterator<Producto> iter = mProducto.iterator();
        while(iter.hasNext()) {
            Producto p = iter.next();
            System.out.printf("Codigo: %d Descripcion: %s Precio: %5.2
f\n",
                              p.getCodigo(), p.getDescripcion(), p.get
Precio());
        }
    }

    public void mostrar() {
        System.out.printf("Tipo: %c Numero: %d Fecha: %d%d%d\n",
                          getTipo(), getNumero(), getFecha().getDia(),
                          getFecha().getMes(), getFecha().getanio());
        System.out.printf("Cliente: \n");
        System.out.printf("Codigo: %d RazonSocial: %s\n",
                          mCliente.getCodigo(), mCliente.getRazonSocia
l());
        System.out.printf("Productos: \n");
        mostrarProducto();
        System.out.printf("Total: %5.2f \n", getTotal());
    }

// Otros getters y setters
}
```

La clase `Factura` es la más compleja y demuestra varios conceptos importantes:

1. **Herencia**: Extiende `Comprobante`.

2. **Agregación**: Contiene un `Cliente` y una lista de `Producto`.
3. **Colecciones**: Utiliza `ArrayList` para almacenar múltiples productos.
4. **Polimorfismo**: A través de la herencia de `Comprobante`.
5. **Encapsulación**: Maneja internamente la lista de productos y el cálculo del total.

El método `agregarProducto()` demuestra cómo la factura mantiene su consistencia interna al actualizar el total automáticamente. El método `mostrar()` proporciona una representación textual completa de la factura, incluyendo todos los objetos agregados.

Análisis de la Clase Principal `AppAgregacion`

```
java
Copy
public class AppAgregacion {
    public static void main(String[] args) throws Exception {
        Fecha hoy = new Fecha(20, 10, 2011);
        Producto pro1 = new Producto(1, "Marron", (float)8.5);
        Producto pro2 = new Producto(2, "Media Luna", 2);
        Cliente cli = new Cliente(1, "Juana");
        Factura f1 = new Factura('F', 1, hoy, cli);

        f1.agregarProducto(pro1);
        f1.agregarProducto(pro2);
        f1.mostrar();
    }
}
```

Esta clase principal demuestra:

1. **Creación de objetos**: Instancia objetos de todas las clases del sistema.
2. **Composición**: Crea una `Factura` utilizando objetos `Fecha` y `Cliente`.
3. **Manipulación**: Agrega `Producto`s a la `Factura`.
4. **Visualización**: Utiliza el método `mostrar()` para presentar todos los datos.

Este código de prueba es efectivo para demostrar la funcionalidad básica del sistema, pero en una aplicación real, se beneficiaría de manejo de excepciones y validación de entrada más robusta.

Resultados

La ejecución del programa demuestra la correcta implementación de la agregación:

- Los objetos se componen de manera lógica y coherente.
- La factura puede manejar múltiples productos y un cliente.
- La información se puede mostrar de manera estructurada.

Conclusiones

El proyecto "Comprobantes" demuestra eficazmente la implementación y los beneficios de la agregación en Java. La estructura de clases permite una composición flexible y lógica de objetos, modelando de manera efectiva un sistema de facturación. Esta implementación no solo facilita la reutilización de código, sino que también proporciona una base sólida para futuras expansiones del sistema.

La capacidad de componer objetos complejos a partir de componentes más simples, como se muestra en la clase `Factura`, resalta la flexibilidad y potencia de la agregación en el diseño orientado a objetos. Sin embargo, es crucial considerar cuidadosamente cómo se gestionan las relaciones entre objetos para mantener la integridad y consistencia del sistema.