BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

INGENIERÍA DE SOFTWARE

DRA. ETELVINA ARCHUNDIA SIERRA

DOCUMENTO DEL PROYECTO: NOCIONES BÁSICAS DE LA ING DE SOFTWARE.

ALUMNOS:

BARBOSA CARRILO, ANTONIO A

ESPINOSA VELAZQUEZ, LUIS A

MATEOS ROMERO, PEDRO

NORIEGA SILVESTRE, JESUS A

RUBIO MARTINEZ, JEDUS D

TALAVERA SANDOVAL, IVÁN

OTOÑO 2021

30-08-2021

EJERCICIOS DE PATRONES DE DISEÑO: DECORADORA, FÁBRICA, FÁBRICA DE MÉTODOS

Decoradora

Es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Ejemplo

```
// La interfaz de componente define operaciones que los
// decoradores pueden alterar.
interface DataSource is
    method writeData(data)
    method readData():data

// Los componentes concretos proporcionan implementaciones por
// defecto para las operaciones. En un programa puede haber
// muchas variaciones de estas clases.
class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escribe datos en el archivo.

    method readData():data is
        // Lee datos del archivo.

// La clase decoradora base sigue la misma interfaz que los
// demás componentes. El principal propósito de esta clase es
// definir la interfaz de encapsulación para todos los
// decoradores concretos. La implementación por defecto del
// código de encapsulación puede incluir un campo para almacenar
// un componente envuelto y los medios para inicializarlo.
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    // La decoradora base simplemente delega todo el trabajo al
    // componente envuelto. En los decoradores concretos se
    // pueden añadir comportamientos adicionales.
    method writeData(data) is
        wrappee.writeData(data)

    // Los decoradores concretos pueden invocar la
    // implementación padre de la operación en lugar de invocar
    // directamente al objeto envuelto. Esta solución simplifica
```

```
    // la extensión de las clases decoradoras.
    method readData():data is
        return wrappee.readData()

// Los decoradores concretos deben invocar métodos en el objeto
// envuelto, pero pueden añadir algo de su parte al resultado.
// Los decoradores pueden ejecutar el comportamiento añadido
// antes o después de la llamada a un objeto envuelto.
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encripta los datos pasados.
        // 2. Pasa los datos encriptados al método writeData
        // (escribirDatos) del objeto envuelto.

    method readData():data is
        // 1. Obtiene datos del método readData (leerDatos) del
        // objeto envuelto.
        // 2. Intenta descifrarlo si está encriptado.
        // 3. Devuelve el resultado.

// Puedes envolver objetos en varias capas de decoradores.
class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Comprime los datos pasados.
        // 2. Pasa los datos comprimidos al método writeData del
        // objeto envuelto.

    method readData():data is
        // 1. Obtiene datos del método readData del objeto
        // envuelto.
        // 2. Intenta descomprimirlo si está comprimido.
        // 3. Devuelve el resultado.

// Opción 1. Un ejemplo sencillo del montaje de un decorador.
class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)
        // El archivo objetivo se ha escrito con datos sin
        // formato.
        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)
        // El archivo objetivo se ha escrito con datos
        // comprimidos.

        source = new EncryptionDecorator(source)
        // La variable fuente ahora contiene esto:
        // Cifrado > Compresión > FileDataSource
        source.writeData(salaryRecords)
```

```
        // El archivo se ha escrito con datos comprimidos y
        // encriptados.

// Opción 2. El código cliente que utiliza una fuente externa de
// datos. Los objetos SalaryManager no conocen ni se preocupan
// por las especificaciones del almacenamiento de datos.
// Trabajan con una fuente de datos preconfigurada recibida del
// configurador de la aplicación.
class SalaryManager is
    field source: DataSource

    constructor SalaryManager(source: DataSource) { ... }

    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)
    // ...Otros métodos útiles...

// La aplicación puede montar distintas pilas de decoradores
// durante el tiempo de ejecución, dependiendo de la
// configuración o el entorno.
class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
```

Fábrica

Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas. Lo primero que sugiere es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos. Después podemos hacer que todas las variantes de los productos sigan esas interfaces.

Ejemplo

```
// La interfaz fábrica abstracta declara un grupo de métodos que
// devuelven distintos productos abstractos. Estos productos se
// denominan familia y están relacionados por un tema o concepto
// de alto nivel.
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox

// Las fábricas concretas producen una familia de productos que
// pertenecen a una única variante. La fábrica garantiza que los
// productos resultantes sean compatibles.
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// Cada fábrica concreta tiene una variante de producto
// correspondiente.
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()

// Cada producto individual de una familia de productos debe
// tener una interfaz base. Todas las variantes del producto
// deben implementar esta interfaz.
interface Button is
    method paint()

// Los productos concretos son creados por las fábricas
// concretas correspondientes.
class WinButton implements Button is
    method paint() is
        // Representa un botón en estilo Windows.

class MacButton implements Button is
    method paint() is
        // Representa un botón en estilo macOS.

// Aquí está la interfaz base de otro producto. Todos los
// productos pueden interactuar entre sí, pero sólo entre
// productos de la misma variante concreta es posible una
// interacción adecuada.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
```

```
    method paint() is
        // Representa una casilla en estilo Windows.

class MacCheckbox implements Checkbox is
    method paint() is
        // Representa una casilla en estilo macOS.

// El código cliente funciona con fábricas y productos
// únicamente a través de tipos abstractos: GUIFactory, Button y
// Checkbox. Esto te permite pasar cualquier subclase fábrica o
// producto al código cliente sin descomponerlo.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()

// La aplicación elige el tipo de fábrica dependiendo de la
// configuración actual o de los ajustes del entorno y la crea
// durante el tiempo de ejecución (normalmente en la etapa de
// inicialización).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")

        Application app = new Application(factory)
```

Fábrica de métodos

Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán. El patrón sugiere que, en lugar de llamar al operador “new” para construir objetos directamente, se invoque a un método fábrica especial.

Ejemplo

```
// La clase creadora declara el método fábrica que debe devolver
// un objeto de una clase de producto. Normalmente, las
// subclases de la creadora proporcionan la implementación de
// este método.
class Dialog is
    // La creadora también puede proporcionar cierta
    // implementación por defecto del método fábrica.
    abstract method createButton():Button

    // Observa que, a pesar de su nombre, la principal
    // responsabilidad de la creadora no es crear productos.
    // Normalmente contiene cierta lógica de negocio que depende
    // de los objetos de producto devueltos por el método
    // fábrica. Las subclases pueden cambiar indirectamente esa
    // lógica de negocio sobrescribiendo el método fábrica y
    // devolviendo desde él un tipo diferente de producto.
    method render() is
        // Invoca el método fábrica para crear un objeto de
        // producto.
        Button okButton = createButton()
        // Ahora utiliza el producto.
        okButton.onClick(closeDialog)
        okButton.render()

// Los creadores concretos sobrescriben el método fábrica para
// cambiar el tipo de producto resultante.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()


// La interfaz de producto declara las operaciones que todos los
// productos concretos deben implementar.
interface Button is
    method render()
    method onClick(f)

// Los productos concretos proporcionan varias implementaciones
// de la interfaz de producto.

class WindowsButton implements Button is
    method render(a, b) is
        // Representa un botón en estilo Windows.
```

```
    method onClick(f) is
        // Vincula un evento clic de OS nativo.

class HTMLButton implements Button is
    method render(a, b) is
        // Devuelve una representación HTML de un botón.
    method onClick(f) is
        // Vincula un evento clic de navegador web.

class Application is
    field dialog: Dialog

    // La aplicación elige un tipo de creador dependiendo de la
    // configuración actual o los ajustes del entorno.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // El código cliente funciona con una instancia de un
    // creador concreto, aunque a través de su interfaz base.
    // Siempre y cuando el cliente siga funcionando con el
    // creador a través de la interfaz base, puedes pasarle
    // cualquier subclase del creador.
    method main() is
        this.initialize()
        dialog.render()
```