Actividad 1 - Proceso de diseño de software

Facultad de Ingeniería,
Corporación Universitaria Iberoamericana
Fundamentos de Diseño

Lady Tatiana Peña Cruz
2020

Elaborar un documento que contenga los siguientes ítems:

Un cuadro de resumen con los principios del diseño de software, detallando las palabras claves. Un cuadro comparativo con la descripción de los patrones GRASP y contrastarlos con los principios SOLID.

Los principios básicos de diseño hacen posible que el ingeniero del software navegue por el proceso de diseño [Pressman, 2002]

- En el proceso de diseño no deberá utilizarse “orejeras”
- El diseño deberá poderse rastrear hasta el modelo de análisis
- El diseño no deberá inventar nada que ya esté inventado
- El diseño deberá minimizar la distancia intelectual entre el software y el problema, como si de misma vida real se tratara
- El diseño deberá presentar uniformidad e integración
- El diseño deberá estructurarse para admitir cambios
- El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes
- El diseño no es escribir código y escribir código no es diseñar
- El diseño deberá evaluarse en función de la calidad mientras que se va creando, no después de terminarlo
- El diseño deberá revisarse para minimizar los errores conceptuales (semánticos)

Cuadro comparativo con la descripción de los patrones GRASP y contrastarlos con los principios SOLID

Patrones GRASP	**Principios SOLID**
Los patrones GRASP describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones.	Todo desarrollo, antes o después, se ve sometido a cambios sobre la funcionalidad inicial o simplemente se requiere funcionalidad nueva.
GRASP es un acrónimo que significa General Responsibility Asignment Software Patterns (patrones generales de	Estos principios nos proporcionan unas bases para la construcción de aplicaciones mucho más sólidas y robustas. Permiten

software para asignar responsabilidades).

El nombre se eligió para indicar la importancia de captar (grasping)

Estos principios, si se quiere diseñar eficazmente el software orientado a objetos. Estos patrones son:

- Experto
- Creador
- Bajo Acoplamiento
- Alta Cohesión
- Controlador

Patron Experto

Problema

¿Cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño orientado a objetos?

Un modelo de clase puede definir docenas y hasta cientos de clases de software, y una aplicación tal vez requiera el cumplimiento de cientos o miles de responsabilidades.

Si estas se asignan en forma adecuada, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y se nos presenta la oportunidad de que los cambios y la evolución del software tengan un mínimo impacto en el código ya desarrollado tratando de evitar que lo que funciona deje de funcionar y por ello que el coste del mantenimiento sea mucho menor.

A continuación pasamos a ver SOLID que es un acrónimo de 5 principios básicos en el diseño y programación orientada a objetos.

Single responsibility (Principio de responsabilidad única)

Este principio dice que cada clase tiene que tener una responsabilidad única y concreta, como dice Rober C. Martín “Una clase debería tener una y sólo una razón para cambiar”.

Es común que cuando empezamos a desarrollar se acabe tomando decisiones como meter en una clase un método a causa de que esa clase lo utiliza, el problema llega cuando más adelante ese método lo necesitamos usar en otras clases y vemos que pierde coherencia.

Supongamos que tenemos que realizar una

reutilizar los componentes en futuras aplicaciones.

Solución

Asignar una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir la responsabilidad.

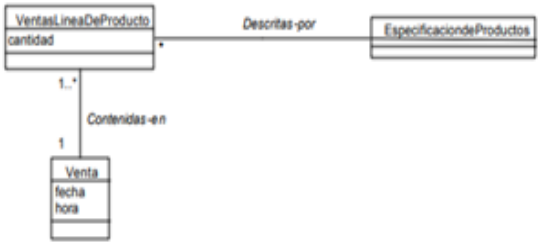
Ejemplo

En la aplicación del punto de venta, alguna clase necesita conocer el gran total de la venta. Comience asignando las responsabilidades con una definición clara de ellas. A partir de esta recomendación se

plantea la pregunta:

¿Quién es el responsable de conocer el gran total de la venta?

Desde el punto de vista del patrón Experto, deberíamos buscar la clase de objetos que posee la información necesaria para calcular el total.



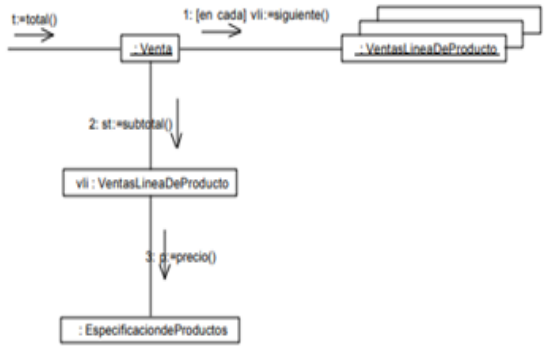
aplicación para registrar las ofertas presentadas y las ventas cerradas. Se puede crear una clase vendedor con un método “GenerarOferta” que registrará las ofertas presentadas y otro “CerrarVenta” que registrará las ofertas ganadas.

```
public class Venta
{
    public void GenerarOferta() { }

    public void CerrarVenta() { }
}
```

El ejemplo, pude parecer correcto, pero en realidad estamos mezclando 2 conceptos (oferta y venta). La forma en que se realiza una oferta puede variar por diversos motivos, lo mismo ocurre con el procedimiento de venta que también puede cambiar, lo cual implicará modificar la clase por 2 motivos diferentes.

En conclusión, para cumplir con la responsabilidad de conocer y dar el total de la venta, se asignaron tres responsabilidades a las tres clases de objeto así:

Clase	Responsabilidad
Venta	conoce el total de la venta
VentasLineadeProducto	conoce el subtotal de la línea de producto
EspecificaciondeProducto	conoce el precio del producto



Experto es un patrón que se usa más que cualquier otro al asignar responsabilidades; es un principio básico que suele ser útil en el diseño orientado a objetos.

Nótese, que el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos.

Beneficios

Se conserva el encapsulamiento, ya que los

```
public class Oferta
{
    public void GenerarOferta() { }
}

public class Venta
{
    public void CerrarVenta(Oferta oferta) { }
}
```

Después de hacer las modificaciones, vemos que si ahora cambia la forma de realizar una oferta no impacta sobre la venta y si cambia la forma de cerrar la venta no impacta sobre la oferta.

objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un bajo acoplamiento, lo que favorece al hecho de tener sistemas más robustos y de fácil mantenimiento.

El comportamiento se distribuye entre las clases que cuentan con la información requerida, alentando con ello definiciones de clase “sencillas” y más cohesivas que son más fáciles de comprender y de mantener. Así se brinda soporte a una alta cohesión.

Patron Creador

Problema

¿Quién debería ser responsable de crear una nueva instancia de alguna clase?

La creación de objetos es una de las actividades más frecuentes en un sistema orientado a objetos.

En consecuencia, conviene contar con un principio general para asignar las Responsabilidades concernientes a ella.

El diseño, bien asignado, puede soportar un bajo acoplamiento, una mayor

Open-Close (Principio de abierto-cerrado)

Lo que dice este principio es que no se debe modificar el código de una clase o entidad, sino que estas deben de ser extendidas y para que esto se cumpla, nuestro código debe estar bien diseñado. La forma más común para cumplir con este principio es el uso de la herencia y/o el de las interfaces.

Cumplir el principio de responsabilidad única ayuda a que este otro principio también se cumpla, pero no significa que cumpliendo uno se cumpla el otro.

claridad, el encapsulamiento y la reutilizabilidad.

El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos, tarea muy frecuente en los sistemas orientados a objetos.

El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento.

Al escogerlo como creador, se da soporte al bajo acoplamiento.

Solución

Asignarle a la clase B la responsabilidad de crear una instancia de clase A en uno de los siguientes casos:

B agrega los objetos A.

B contiene los objetos A.

B registra las instancias de los objetos A o

B utiliza especialmente los objetos A.

B tiene los datos de inicialización que serán transmitidos a A cuando este objeto sea creado (así que B es un Experto respecto a La creación de A). B es un creador de los objetos A.

Sigamos con el ejemplo anterior, supongamos ahora que vamos a realizar la oferta y que dependiendo del tipo de servicio tendremos que generar una estructura diferente en cada caso. Añadiremos una propiedad de tipo enumerado en la clase Oferta que nos indique el tipo de servicio y un método por cada servicio que genere una estructura u otra, cuando realicemos la oferta tendremos que comprobar el tipo y en función de este llamar a uno u otro método.

```
public enum Servicios
{
    OutSourcing,
    Desarrollo
}

public class Oferta
{
    public Servicios Servicio { get; set; }

    public void GenerarOfertaOutSourcing() { }

    public void GenerarOfertaDesarrollo() { }
}

public class Main
{
    public void RealizarOferta(Oferta oferta)
    {
        switch (oferta.Servicio)
        {
            case Servicios.OutSourcing:
                oferta.GenerarOfertaOutSourcing();
                break;
            case Servicios.Desarrollo:
                oferta.GenerarOfertaDesarrollo();
                break;
        }
    }
}
```

En este ejemplo, si queremos ofrecer un nuevo servicio nos vemos obligados a modificar la clase Oferta y el método “RealizarOferta”, por lo que estamos

Si existe más de una opción, prefiera la clase B que agregue o contenga la clase A.

Ejemplo

En la aplicación del punto de venta, ¿quién debería encargarse de crear una instancia VentasLineadeProducto?

Desde el punto de vista del patrón Creador, deberíamos buscar una clase que agregue, contenga y realice otras operaciones sobre este tipo de instancias.

Una Venta contiene (en realidad, agrega) muchos objetos VentasLineadeProducto;

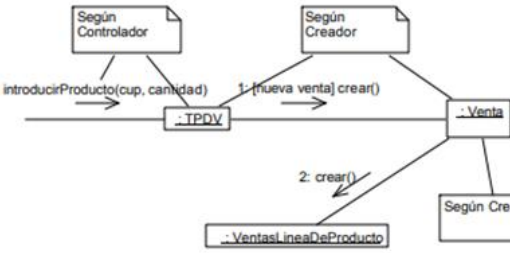
por ello, el patrón Creador sugiere que Venta es idónea para asumir la Responsabilidad de crear las instancias VentasLineadeProducto.

Esta asignación de responsabilidades requiere definir en Venta un método de hacer-LineadeProducto.

incumpliendo el principio abierto-cerrado. En el siguiente ejemplo mostramos como se podría solucionar.

```
public abstract class Oferta
{
    public abstract void GenerarOferta();
}

public class OutSourcing : Oferta
{
    public override void GenerarOferta() { }
}

public class Desarrollo : Oferta
{
    public override void GenerarOferta() { }
}

public class Main
{
    public void RealizarOferta(Oferta oferta)
    {
        oferta.GenerarOferta();
    }
}
```

Como podemos ver en el ejemplo, al crear por cada tipo del enumerado una clase que herede de Oferta e implemente su método “GenerarOferta”, conseguimos simplificar la funcionalidad de forma que no sea necesario realizar modificaciones si añadimos un nuevo servicio, ya que bastará con crear una nueva clase que herede de Oferta e implemente el método. En el siguiente ejemplo añadimos el servicio de consultoría.




```
public abstract class Oferta
{
    public abstract void GenerarOferta();
}
public class OutSourcing : Oferta
{
    public override void GenerarOferta() { }
}
public class Consultoria : Oferta
{
    public override void GenerarOferta() { }
}
public class Desarrollo : Oferta
{
    public override void GenerarOferta() { }
}
public class Main
{
    public void RealizarOferta(Oferta oferta)
    {
        oferta.GenerarOferta();
    }
}
```

En ocasiones encontramos un patrón creador buscando la clase con los datos de inicialización que serán transferidos durante la creación.

Éste es en realidad un ejemplo del patrón Experto.

Los datos de inicialización se transmiten durante la creación a través de algún método de inicialización, como un constructor en java que cuenta con parámetros.

Patron Bajo Acoplamiento

Problema

¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?

El acoplamiento es una medida de la fuerza con que una clase está conectada a otras clases, con que las conoce y con que recurre a ellas.

Liskov Substitution (Principio de Sustitución de Liskov)

Este principio recibe su nombre por su creadora Barbara Liskov. Viene a decir que una clase derivada de otra debe poder ser sustituida por su clase base y debemos garantizar que los métodos de la primera no provoquen un mal funcionamiento de los métodos de la clase base.

Acoplamiento bajo significa que una clase no depende de muchas clases.

Acoplamiento alto significa que una clase recurre a muchas otras clases. Esto presenta los siguientes problemas:

Los cambios de las clases afines ocasionan cambios locales.

Difíciles de entender cuando están aisladas.

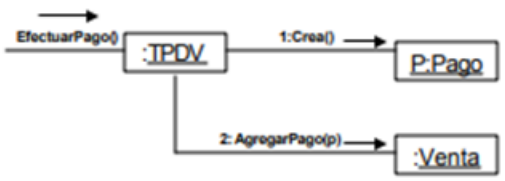
Difíciles de reutilizar puesto que dependen de otras clases.

Solución

Asignar una responsabilidad para mantener bajo acoplamiento.

Ejemplo: En el caso del punto de ventas se tienen tres clases Pago, TPDV y Venta y se quiere crear una instancia de Pago y asociarla a Venta. ¿Qué clase es la responsable de realizarlo?

Según el patrón experto la Clase TPDV deberá hacerlo



Según el patrón de Bajo Acoplamiento la relación debería ser de la siguiente

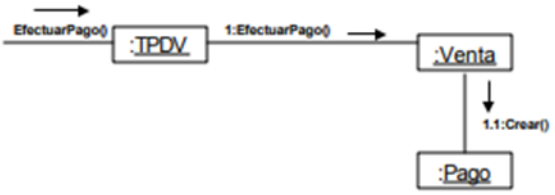
Vamos a utilizar un ejemplo muy recurrido para explicar este principio, el del cuadrado y el rectángulo. Un cuadro no es más que un rectángulo con todos los lados iguales, intentemos llevar esto a la programación y calcular su área.

```
public class Rectangulo
{

    private int ancho;
    private int alto;

    public int getAncho()
    {
        return ancho;
    }

    public virtual void setAncho(int ancho)
    {
        this.ancho = ancho;
    }

    public int getAlto()
    {
        return alto;
    }

    public virtual void setAlto(int alto)
    {
        this.alto = alto;
    }

    public int calcularArea()
    {
        return ancho * alto;
    }
}

public class Cuadrado : Rectangulo
{
    public override void setAncho(int ancho)
    {
        base.setAncho(ancho);
        base.setAlto(ancho);
    }

    public override void setAlto(int alto)
    {
        base.setAncho(alto);
        base.setAlto(alto);
    }
}
```

Hemos creado una clase “Rectángulo” con unos métodos para obtener ancho y alto, otros para modificar el ancho y el alto, y un método que calcula el área.

Hemos creado también una clase “Cuadrado” que hereda de “Rectángulo”

manera:



Esta última asociación es mejor dado que Venta realiza la creación del Pago y no TPDV por lo tanto se reduce la dependencia de este último con el resto de las clases.

El grado de acoplamiento no puede considerarse aisladamente de otros principios como Experto y Alta Cohesión. Sin embargo, es un factor a considerar cuando se intente mejorar el diseño.

En los lenguajes OO como C++ y JAVA las formas comunes de acoplamiento de TipoX a TipoY son las siguientes:

TipoX posee un atributo (miembro de datos o variable de instancia) que se refiere a una instancia TipoY o al propio TipoY.

TipoX tiene un método que a toda costa referencia una instancia de TipoY o incluso el propio TipoY .Suele incluirse un parámetro o una variable y sobrescribe los métodos que cambian el ancho y alto para que mantengan una relación.

```
[TestMethod]
public void testArea()
{
    Rectangulo r = new Rectangulo();
    r.setAncho(5);
    r.setAlto(4);
    assertEquals(20, r.calcularArea());
}
```

El siguiente método lo usamos para testear nuestras clases. Si nos fijamos siempre que se pase un rectángulo funcionará correctamente, pero si cambiamos el rectángulo por el cuadrado no funcionará y por ello incumple el principio de Liskov.

```
public class Rectangulo
{
    public int ancho;
    public int alto;

    public Rectangulo(int ancho, int alto)
    {
        this.ancho = ancho;
        this.alto = alto;
    }

    public int calcularArea()
    {
        return ancho * alto;
    }
}

public class Cuadrado : Rectangulo
{
    public Cuadrado(int lado) : base(lado,lado) { }
}
```

Como se puede ver, hemos cambiado las clases para que sus constructores reciban ancho y alto por parámetros, el

local de TipoY o bien el objeto devuelto de un mensaje es una instancia de TipoY.

TipoX es una subclase directa o indirecta del TipoY.

TipoY es una interfaz y TipoX la implementa.

Beneficios:

No se afectan por cambios de otros componentes

Fáciles de entender por separado

Fáciles de reutilizar.

Patron Alta Cohesión

Problema

¿Cómo mantener la complejidad dentro de límites manejables?

La cohesión es una medida de cuán relacionadas y enfocadas están las Responsabilidades de una clase.

Una alta cohesión caracteriza a las clases con responsabilidades estrechamente relacionadas que no realicen un trabajo enorme.

Una baja cohesión hace muchas cosas no afines o realiza trabajo excesivo. Esto presenta los siguientes problemas:

Son difíciles de comprender

lado en caso del cuadrado. De esta forma estamos obligando en la definición del objeto a indicar el valor que tendrán sus lados, haciendo su uso más intuitivo y evitando que haya operaciones por detrás que nos lleven a confusión.

Interface Segregation (Principio de Segregación de Interfaces)

Mejor crear muchas interfaces que contengan pocos métodos, que crear pocas interfaces que definan demasiados métodos. El motivo de este principio es que ninguna clase debe de estar obligada a implementar métodos que no necesita, por ello es preferible crear varias interfaces que agrupen funcionalidad común, a englobar gran parte de esa funcionalidad en unas pocas interfaces y encontrarnos más adelante que necesitamos implementar una interface que nos obliga a implementar métodos que no

Difíciles de reutilizar

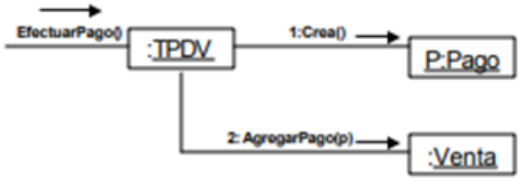
Difíciles de conservar

Las afectan constantemente los cambios.

Solución

Asignar una responsabilidad de modo que la cohesión siga siendo alta

Ejemplo: En el caso del punto de ventas se tienen tres clases Pago, TPDV y Venta y se quiere crear una instancia de Pago y asociarla a Venta. Según el principio del patrón Creador la clase TPDV debe ser la encargada de realizar el pago.



¿Qué pasa si el sistema tiene 50 operaciones, todas recibidas por la Clase TPDV?

La clase se iría saturando con tareas y terminaría perdiendo la cohesión

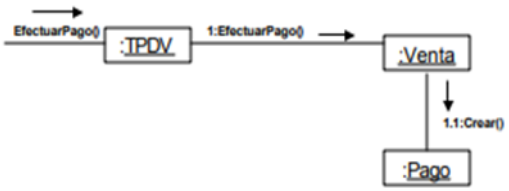
Un mejor diseño de lo anterior sería:

necesitamos.

En el siguiente ejemplo volvemos a nuestro proyecto de las ofertas.

```
public interface IOferta
{
    void GenerarOferta();
    void EstablecerDesplazamiento();
    void EstablecerTarifa();
}

public class OutSourcing : IOferta
{
    public void GenerarOferta() { }

    public void EstablecerDesplazamiento() { }

    public void EstablecerTarifa() { }
}

public class Proyecto : IOferta
{
    public void GenerarOferta() { }

    public void EstablecerDesplazamiento()
    {
        throw new NotImplementedException();
    }

    public void EstablecerTarifa() { }
}
```

Esta vez, tendremos una interface “IOferta” que define una serie de métodos que se usan en las ofertas. Si nos fijamos, cuanto heredamos de esa interface en la clase “Proyecto” nos vemos obligados a implementar el método “EstablecerDesplazamiento”, que nos viene bien para el outsourcing, pero en proyectos que se van a desarrollar internamente no lo necesitamos y si lo usamos devolverá una excepción.



Este diseño delega a Venta la responsabilidad de crear el pago.

Este diseño es conveniente ya que da soporte a una alta cohesión y a un bajo acoplamiento.

En la práctica, el nivel de cohesión no puede ser considerado independiente de los otros patrones y principios (e.g. Patrones "Experto" y "Bajo Acoplamiento").

Beneficios:

Mejoran la claridad y facilidad con que se entiende el diseño

Se simplifica el mantenimiento y las mejoras de funcionalidad

A menudo se genera un bajo acoplamiento

Soporta mayor capacidad de reutilización.

Algunos escenarios:

Muy baja cohesión: Una clase es la única responsable de muchas cosas en áreas funcionales heterogéneas.

Baja cohesión: Una clase tiene la

```
public interface IOferta
{
    void GenerarOferta();
    void EstablecerTarifa();
}
public interface IOutSourcing
{
    void EstablecerDesplazamiento();
}
public class OutSourcing : IOferta, IOutSourcing
{
    public void GenerarOferta() { }

    public void EstablecerDesplazamiento() { }

    public void EstablecerTarifa() { }
}
public class Proyecto : IOferta
{
    public void GenerarOferta() { }

    public void EstablecerTarifa() { }
}
```

Lo que hemos hecho para solucionarlo es crear una nueva interface que defina el método "EstablecerDesplazamiento" e implementarla solo donde se necesita. Ahora no tenemos métodos implementados que no necesitamos usar.

responsabilidad exclusiva de una tarea Compleja dentro de un área funcional.

Alta cohesión: Una clase tiene responsabilidades moderadas en un área Funcional y colabora con las otras para llevar a cabo las tareas.

Cohesión moderada: Una clase tiene peso ligero y responsabilidades exclusivas en unas cuántas áreas que están relacionadas lógicamente con el concepto de clase pero no entre ellas.

Patron Controlador

Problema:

¿Quién debería encargarse de atender un evento del sistema?

Un evento del sistema es un evento de alto nivel generado por un actor externo; es un evento de entrada externa. Se asocia a Operaciones del sistema: las que emite en respuesta a los eventos del sistema.

Por ejemplo, cuando un cajero que usa un sistema de terminal en el punto de venta oprime el botón "Terminar Venta", está generando un evento sistémico que indica que “la venta ha

Dependency Inversion (Principio de Inversión de Dependencia)

Este principio busca que no exista un alto acoplamiento en las aplicaciones, ya que ello repercute en un difícil mantenimiento.

El principio quiere decir que las clases de alto nivel no tienen que depender de otras de bajo nivel, sino que ambas dependan de abstracciones, así como que las abstracciones no deben depender de los detalles, sino al contrario.

Imaginemos que tenemos que generar una oferta y que dependiendo del su tipo se

terminado”.

Un Controlador es un objeto de interfaz no destinada al usuario que se encarga de manejar un evento del sistema. Define además el método de su operación.

Solución

Asignar la responsabilidad del manejo de un mensaje de los eventos de un sistema a una clase que represente una de las siguientes

opciones:

El “sistema” global (controlador de fachada).

La empresa u organización global (controlador de fachada).

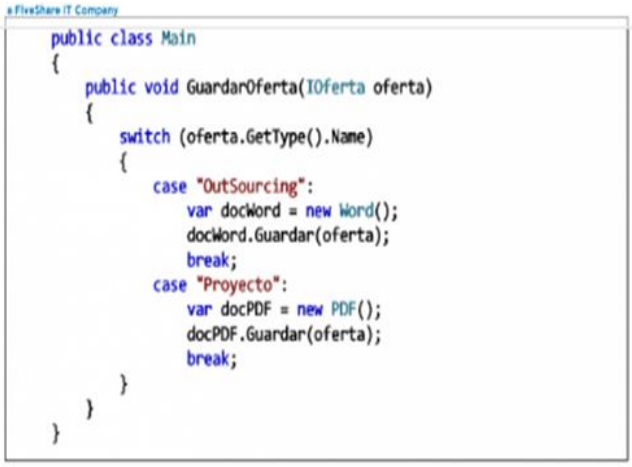
Algo en el mundo real que es activo (por ejemplo, el papel de una persona) y que pueda participar en la tarea (controlador de tareas).

un manejador artificial de todos los eventos del sistema de un caso de uso, generalmente denominados “Manejador<NombreCasodeUso>” (Controlador de casos de uso).

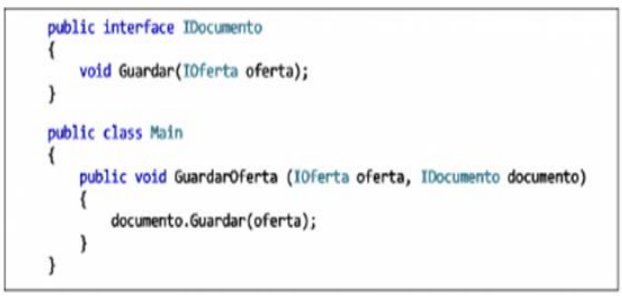
Ejemplo:

En la aplicación del punto de venta se dan

generará con una estructura u otra.

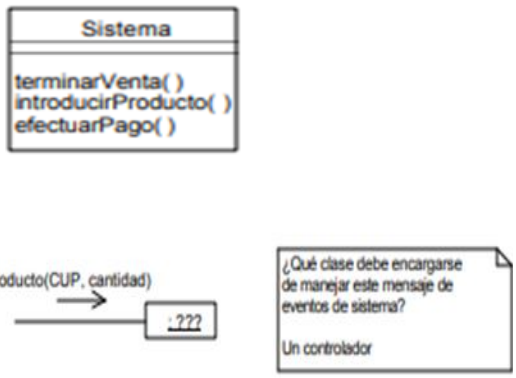


En este ejemplo disponemos de un método principal que recibe una oferta y en base al tipo de esta decidimos se guarda en Word o en PDF. El problema que encontramos es que hay un fuerte acoplamiento y dependencias entre las clases. Si más adelante se requiere que una oferta concreta se guarde en ambos formatos, nos vemos obligados a cambiar toda la lógica del método.



varias operaciones del sistema, como se advierte en la figura.

¿Quién debería ser el controlador de eventos sistémicos como introducirProducto y terminarVenta?



De acuerdo con el patrón Controlador, disponemos de las siguientes opciones:

representa el “sistema” global TPDV

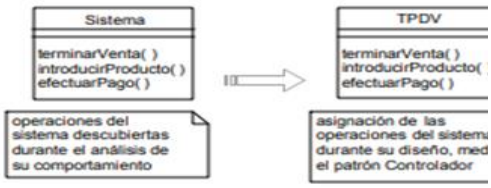
representa la empresa u organización global Tienda

representa algo en el mundo real que está activo (por ejemplo, el papel de una persona) y que puede intervenir en la tarea Cajero

representa un manejador artificial de todas las Operaciones del sistema de un caso de uso. ManejadordeComprarProductos

En la decisión de cuál de las cuatro clases es

el controlador más apropiado influyen también otros factores como la cohesión y el acoplamiento.



Explicación: La mayor parte de los sistemas reciben eventos de entrada externa, los cuales generalmente incluyen una interfaz gráfica para el usuario (IGU) operado por una persona.

La misma clase controlador debería utilizarse con todos los eventos sistémicos de un caso de uso, de modo que podamos conservar la Información referente al estado del caso.

Esta información será útil - por ejemplo - para identificar los eventos del sistema fuera de secuencia (entre ellos, una operación efectuarPago antes de terminarVenta).

Pueden emplearse varios controladores en los casos de Uso.

Un defecto frecuente al diseñar controladores consiste en asignarles demasiada responsabilidad. Normalmente un controlador

debería delegar a otros objetos el trabajo que ha de realizarse mientras coordina la actividad.

- “Manejador artificial de casos de uso” - un controlador para cada caso.
  - Adviértase que éste no es un objeto del dominio, es un concepto artificial, cuyo fin es dar soporte al sistema (una fabricación pura, en términos de los patrones de GRASP).
  - Por ejemplo, si la aplicación del punto de venta contiene casos de uso como “Comprar Productos” y “Devolver Productos”, habrá una clase ManejadordeComprarProductos y una clase ManejadordeDevolverProductos.
  - Es una alternativa que debe tenerse en cuenta, si el hecho de asignar las responsabilidades en cualquiera de las otras opciones de controlador
  - Genera diseños de baja cohesión o alto acoplamiento. Esto ocurre generalmente cuando un controlador empieza a “saturarse” con

Demasiadas responsabilidades.

Beneficios

Mayor potencial de los componentes reutilizables. Garantiza que la empresa o los procesos de dominio sean manejados por la capa de los objetos del dominio y no por la de la interfaz.

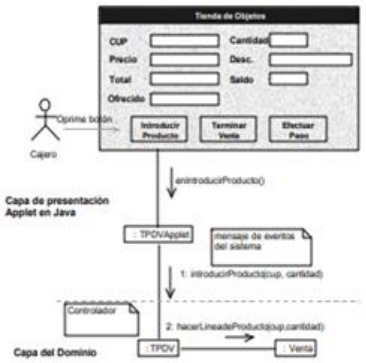
Reflexionar sobre el estado del caso de uso. A veces es necesario asegurarse de que las operaciones del sistema sigan una secuencia

Legal o poder razonar sobre el estado actual de la actividad y las operaciones en el caso de uso subyacente.

Por ejemplo, tal vez deba garantizarse que la operación efectuarPago no ocurra mientras no se concluya la operación terminarVenta. De ser así, esta información sobre el estado ha de capturarse en alguna parte; el controlador es una buena opción, sobre todo si se emplea a lo largo de todo el caso.

Un corolario importante del patrón Controlador es que los objetos de la

interfaz (por ejemplo, objetos de ventanas, applets) y la capa de Presentación no debería encargarse de manejar los eventos del sistema.



- Nótese que la clase *TPDVApplet* - parte de la capa de presentación - transmite un mensaje *introducirProducto* al objeto *TPDV*.
- No intervino en el proceso de la operación, ni la decisión de cómo manejarla, el applet se limitó a delegarla a la capa del dominio.

Referencias Bibliográficas

Lemos. A. (2020). Ingeniería del Software Tema 5: Principios del diseño del software. Recuperado de https://www.academia.edu/9321319/Ingenier%C3%ADa_del_Software_Tema_5_Principios_del_dise%C3%B1o_del_software

Hernández. A. (2020). Patrones de Diseño Contenido? Patrones GRASP? Experto? Creador? Bajo Acoplamiento. Recuperado de https://www.academia.edu/4726489/Patrones_de_Dise%C3%B1o_Contenido_Patrones_GRASP_Experto_Creador_Bajo_Acoplamiento

Rodríguez. L. (2019). Principios básicos del diseño de software (SOLID). Recuperado de https://mvpcluster.com/diseno-de-software-2/