Patrones de Diseño

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Motivación

Motivación

En el diseño de software, es habitual que aparezcan problemas.
Problemas no directamente relacionados con la aplicación, sino con el propio diseño.

Algunos de esos problemas aparecen habitualmente.

Motivación

El programador/diseñador tiene que resolver esos problemas de diseño que se le presenten en el menor tiempo posible.

En un sistema complejo, la solución propuesta puede ser:

- Incorrecta
- Incompleta
- Ineficiente
- Ilegible
- Poco escalable
- ... *Incluye cualquier otro defecto que se te ocurra aquí.*

Motivación

Si ocurre lo siguiente:

- Algunos problemas de diseño aparecen habitualmente.
- En el proceso de diseño software no siempre hay tiempo para encontrar la solución óptima.

¿Por qué no pre-diseñar la solución óptima para esos problemas que aparecen habitualmente?

Patrones de Diseño

Definición

Un patrón de diseño es una solución a un problema de diseño que aparece regularmente en diferentes contextos del desarrollo software.

Contraejemplo

Un patrón de diseño no es una solución a un problema que no es de diseño.
Una lista enlazada o un arbol binario obviamente no son patrones de diseño.

Características

- Debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores.
- Debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

- En 1979 el arquitecto Christopher Alexander aportó al mundo de la arquitectura el libro *The Timeless Way of Building*; en él proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad.
- Publicados en un volumen denominado *A Pattern Language*, son un intento de formalizar y plasmar de una forma práctica generaciones de conocimiento arquitectónico.

- En 1987, Ward Cunningham y Kent Beck usaron varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-ordenador (HCI). Publicaron un artículo titulado *Using Pattern Languages for OO Programs*.
- En la década de 1990 los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro *Design Patterns* escrito por el grupo *Gang of Four* (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes.

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Los patrones de diseño NO pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar al que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable.

Abusar o forzar el uso de los patrones puede ser un error

Nombre

Todos los patrones de diseño documentados se identifican mediante uno o más nombres estándares oficiales. También hay clasificaciones de patrones de diseño. Varios patrones de diseño pueden estar relacionados entre sí, relación que se define mediante sus nombres.

Problema

La característica más importante de un patrón de diseño es el problem que pretende resolver. Tiene que estar bien definido, y sustentado mediante ejemplos de aplicaciones en los que pueda surgir ese problema de diseño.

Solución

Todo patrón de diseño viene acompañado de la solución propuesta, descrita perfectamente y habitualmente acompañada de diagramas.

Implementación

Suele incluirse siempre una descripción de los detalles de implementación para uno u otro lenguaje (la problemática de Java y C++ puede ser distinta), así como código fuente de ejemplo.

Consecuencias

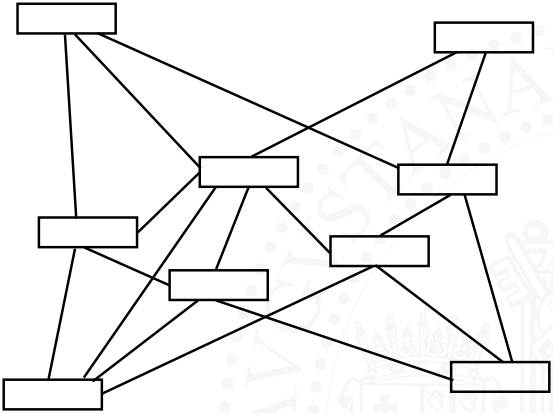
La aplicación de todo patrón de diseño tiene una serie de consecuencias en la aplicación, que pueden ser tanto positivas como negativas.
Puede que un patrón de diseño ofrezca una mayor escalabilidad del código a coste de eficiencia.
Este tipo de comportamientos también deben de estar documentados.

Ejemplos: Facade

Problema

¿Qué ocurre con aplicaciones creadas sin demasiado diseño previo que crecen muy rápido?

Ejemplos: Facade



Problema

¿Qué ocurre con aplicaciones no demasiado bien diseñadas que crecen muy rápido?

- Muchas clases, algunas de ellas puede que redundantes.
- Muchas relaciones entre clases (clases que usan otras, clases que heredan de otras, métodos que devuelven ciertas clases...)
- Modificar una de esas clases puede resultar en el desencadenamiento de una serie de errores en todas las clases que dependen de ella.

¿Solución?

¿Qué se puede hacer en estos casos?

¿Solución?

¿Qué se puede hacer en estos casos?

- Tirar todo el código a la basura y empezar de cero.

¿Solución?

¿Qué se puede hacer en estos casos?

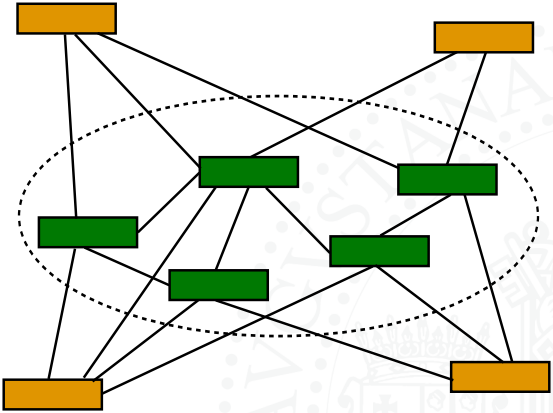
- Tirar todo el código a la basura y empezar de cero.
- Asumir que cada cambio o ampliación en el código sea una locura (código poco mantenible).

¿Solución?

¿Qué se puede hacer en estos casos?

- Tirar todo el código a la basura y empezar de cero.
- Asumir que cada cambio o ampliación en el código sea una locura (código poco mantenible).
- Identificar subsistemas que puedan ser mantenidos y tratados independientemente.

Ejemplos: Facade



Patrón Facade

Paso 1

Se identifican los siguientes grupos de clases:

- Las que pertenecen a un *subsistema*
- Las que utilizan el subsistema, son *clientes* del subsistema.

Se idetifica la funcionalidad global que requieren los *clientes* del subsistema.

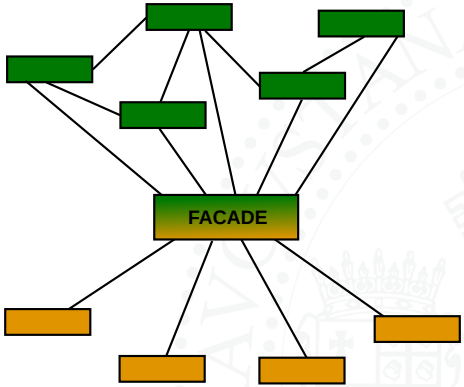
Patrón Facade

Paso 2

Se crea una clase nueva, denominada *Facade*:

- Todas las clases del subsistema, mantienen sus relaciones entre sí y con la clase *Facade*.
- Todas las clases clientes, pasan a relacionarse exclusivamente con la clase *Facade*, que provee la funcionalidad combinada que necesitan de las diferentes clases del subsistema.

Patrón Facade



Patrón Facade

Funcionamiento

1. Una clase cliente envía su petición mediante la invocación de métodos de la clase Facade.
2. El método de la clase Facade obtiene la información necesaria de las clases del subsistema.
3. El método de la clase Facade procesa dicha información.
4. La clase Facade responde a la petición de la clase cliente.

Nótese que no existe comunicación ni invocación directa entre una clase cliente y las clases del subsistema.

Facade = Fachada

Facade: ejemplo

```
1  // Gestiona las reservas de hoteles
2  class ReservaHoteles {
3    ArrayList<Hotel> obtenerHoteles(Ciudad ciudad,Date desde,Date hasta)
4    {
5      // Procesa fechas y devuelve los hoteles libres
6    }
7  }
```

```
1  //Gestiona las reservas de vuelos
2  class ReservaVuelos {
3    ArrayList<Vuelo> obtenerVuelos(Ciudad org,Ciudad dst,Date fecha)
4    {
5      // Devuelve los vuelos en la fecha correspondiente de una ciudad a otra
6    }
7  }
```

Facade: ejemplo

¿Problemas?

¿Qué ocurriría al utilizar ambas clases desde diferentes fragmentos de código (formulario web *vs* app móvil) para organizar un viaje?

Facade: ejemplo

¿Problemas?

¿Qué ocurriría al utilizar ambas clases desde diferentes fragmentos de código (formulario web *vs* app móvil) para organizar un viaje?

- Los dos fragmentos de código necesitan reproducir el mismo código (buscar hoteles y dos vuelos).
- Organizar viajes desde otra parte del código require replicar las mismas invocaciones de métodos.
- Extender la funcionalidad (alquilar coche, visitas guiadas...) implica replicar el código correspondiente en diferentes lugares, con muchas posiblidades de error.

Facade: ejemplo

Idea

La organización de viajes en sí misma es un subsistema.

Podemos utilizar el patrón Facade.

Facade: ejemplo

```
1  //Gestiona los viajes. Es un FACADE
2  class OrganizaViajes {
3    ReservaHoteles reservahoteles;
4    ReservaVuelos  reservavuelos;
5
6    public void obtenerViajes(Ciudad origen, Ciudad destino,
7                              Date ida, Date vuelta)
8    {
9        ArrayList<Vuelo> idas
10             = reservavuelos.obtenerVuelos(origen, destino, ida);
11       ArrayList<Hotel> hoteles
12             = reservahoteles.obtenerHoteles(destino,ida,vuelta);
13       ArrayList<Vuelo> vueltas
14             = reservavuelos.obtenerVuelos(destino, origen, vuelta);
15       //Procesa y devuelve las diferentes posibilidades para el viaje
16   }
17 }
```

Patrón Facade

Ventajas

- Separa las clases cliente de las clases del subsistema. Reduce el número de objetos con los que cada clase cliente tiene que trabajar.
- Abstrae el funcionamiento del subsistema.
- Distribución de un sistema grande en capas y módulos.
- Elimina posibles referencias circulares.
- Permite rediseñar el subsistema completamente sin tocar el resto del sistema.

¿Alguna ventaja más?

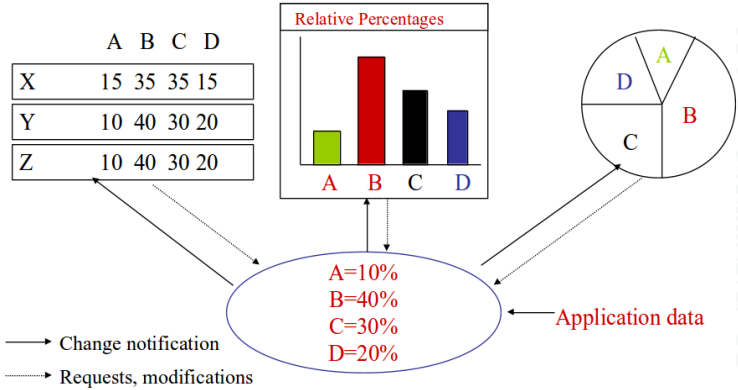
Ejemplos: Observer

Problema

¿Qué ocurre con un dato que se visualiza de diferentes formas?

¿Qué ocurre con un dato que se edita mediante diferentes mecanismos?

Ejemplos: Observer



Problema

¿Qué ocurre con un dato que se visualiza de diferentes formas?
¿Qué ocurre con un dato que se edita mediante diferentes mecanismos?

- Cada vez que uno de los *editores* modifica el dato, debe asegurarse de que su(s) visualización(es) se actualiza(n) correctamente.
- Cada *editor* que añadimos ha de hacer lo mismo.
- Cada *visualizador* que añadimos, debe ser actualizado desde todos los editores.
- Si el sistema es muy dinámico (ventanas que se abren y se cierran...) hay muchos cambios que tener en cuenta.

Ejemplos: Observer

Equipo.java

```
1  //Gestiona el equipo del personaje
2  public class Equipo {
3     public void equipar(Item newitem) // Reemplaza a this.item
4     {
5        if (ventanaEquipo.abierta()) // ¿ Y si hay varias ?
6        {
7        ^^IventanaEquipo.borrar(this.item);
8        ^^IventanaEquipo.pintar(newitem);
9        }
10       modeloPersonaje.borrar(this.item);
11       modeloPersonaje.pintar(newitem);
12       personaje.quitarAtributos(this.item);
13       personaje.ponerAtributos(newitem);
14       this.item = newitem;
15    }
16 }
```

¿Solución?

¿Qué se puede hacer en estos casos?

¿Solución?

¿Qué se puede hacer en estos casos?

- Limitar la funcionalidad: permitir sólo un modo de visualización, y sólo uno de edición.

¿Solución?

¿Qué se puede hacer en estos casos?

- Limitar la funcionalidad: permitir sólo un modo de visualización, y sólo uno de edición.
- Asumir que tenemos en cuenta todos los posibles métodos de actualización y edición del dato correspondiente.

¿Solución?

¿Qué se puede hacer en estos casos?

- Limitar la funcionalidad: permitir sólo un modo de visualización, y sólo uno de edición.
- Asumir que tenemos en cuenta todos los posibles métodos de actualización y edición del dato correspondiente.
- Centralizar la edición y la actualización de todas las potenciales visualizaciones en un sólo método.

Patrón Observer

Paso 1

Encapsular en una sola clase *notificadora*:

- El dato que puede ser editado (acceso privado)
- Una lista de observadores, que voluntariamente pueden *suscribirse* o *desuscribirse* del notificador, con un interfaz común para ser *notificados* de los cambios.
- Un único método para editar el dato correspondiente, que cambia el valor del dato y notifica a todos los observadores suscritos.

Patrón Observer

Paso 2

Cada clase que pudiera editar el dato, deberá:

- Guardar una referencia a la clase notificadora.
- Editar el dato a través del método correspondiente, que a su vez notifica a todos los observadores.

Observer = Observador

Patrón Observer

EquipoObserver.java

```
1  //Implementado por todos los observers
2  interface EquipoObserver {
3     public void alEquipar(Item olditem, Item newitem);
4  }
```

Equipo.java

```
1  class Equipo {
2     void equipar(Item newitem) {
3        notificarEquipar(this.olditem, newitem);
4        this.olditem = newitem;
5     }
6  }
```

Patrón Observer

Equipo.java

```
1  class Equipo {
2     ...
3     ArrayList<EquipoObserver> observers = new ArrayList<>();
4
5     void suscribir(EquipoObserver observer) {
6        if (!observers.contains(observer)) observers.add(observer);
7     }
8     void desuscribir(EquipoObserver observer) {
9          observers.remove(observer);
10    }
11    void notificarEquipar(Item olditem, Item newitem) {
12      for (EquipoObserver o : observers)
13        o.alEquipar(olditem,newitem);
14    }
15 }
```

Patrón Observer

Personaje.java

```
1  class Personaje implements EquipoObserver {
2    Personaje(Equipo equipo)
3    {
4    ^^Iequipo.suscribir(this);
5    }
6    void alEquipar(Item olditem, Item newitem) {
7      this.quitarAtributos(this.olditem);
8      this.ponerAtributos(newitem);
9    }
10 }
```

Patrón Observer

VentanaEquipo.java

```
1  class VentanaEquipo implements EquipoObserver {
2     Equipo equipo;
3     Personaje(Equipo eq) { equipo = eq; }
4     void abrir()  { equipo.suscribir(this); }
5     void cerrar() { equipo.desuscribir(this); }
6     void alEquipar(Item olditem, Item newitem) {
7       this.borrar(olditem);
8       this.pintar(newitem);
9     }
10 }
```

Patrón Observer

Ventajas

- Permite separar los datos de sus diferentes formas de visualización.
- Introduce un comportamiento reactivo en ciertas clases.
- Permite invertir referencias entre objetos.
- Es muy útil para interfaces de usuario y programación orientada a eventos.
- ¿Alguna ventaja más?

Java 8 incluye clases/interfaces llamadas *Observer* y *Observable*.
Java 9 lo reemplaza por java.beans, java.Flow y otros.

Otros ejemplos

Factory = Fábrica

Problema

Trabajando en un banco, hay varios tipos de cuentas:

- **Cuentas corrientes :** Tienen un tipo de interés anual (siendo el mensual el resultado de dividirlo por doce).
- **Plazos fijos :** Vienen determinados por un plazo (número de meses) y un tipo de interés a vencimiento. Antes del plazo, da interés 0 %.
- **Cuenta de nómina :** Se ingresa mensualmente una cantidad fija determinada.
- **Cuenta en divisa :** Dada otra cuenta en otra divisa y dado un factor de cambio entre divisas, calcule la cuenta con el factor de cambio aplicado.

Todas las cuentas estan en una jerarquía de clases. Haz una función para cargar una cuenta en una línea de un fichero con la siguiente estructura:

```
1 CuentaCorriente 1000 0.2
2 PlazoFijo 2000 0.5 12
```

Ejemplo de java

```
1 //Conexion JDBC es un Factory
2 Connection connection = DriverManager.getConnection(
3  "jdbc:oracle:thin:@hendrix-oracle.cps.unizar.es:1521:vicious",
4  user,password);
```

La creación de un objeto es cara:

- Memoria dinámica (operador new, vector flexible)
- Threads
- Operaciones con Sincronización/Locking

Solución:

Object Pool

Conjunto de objetos pre–creados y reciclables.

Hay que diseñar el curso de acción si se agota el *pool*.

A falta de un buen libro...

`http://en.wikipedia.org/wiki/Software_design_pattern`

Conclusiones

- Los patrones de diseño proporcionan soluciones óptimas a problemas comunes de diseño de software, diseñadas por gente muy lista a lo largo de muchos años.
- Facilitan la comunicación. Es más fácil decir *Necesitamos un **Facade** aquí...* que *Estaría bien diseñar una clase intermedia que aislara este subsistema completamente....*
  O *este sistema utiliza un modelo **MVC*** en lugar de *hemos separado los datos de la parte de visualización y...*
- Proporcionan un nuevo nivel de abstracción, permiten hacer un diseño sin llegar niveles de detalle de implementación.

Los patrones de diseño proporcionan soluciones, digamos ideas o algoritmos.

Algunos patrones de diseño pueden implementarse de forma reusable, dependiendo del lenguaje de programación.

¿ Cómo podrías hacerlo ?

Ciertos programadores argumentan que los patrones de diseño existen porque existen defectos o carencias en los lenguajes de programación.

Por ejemplo, el patrón *Observer* no tendría sentido en un lenguaje que fuera automáticamente reactivo (Excel).

Un lenguaje de alto nivel podría incluir patrones de diseño como características del propio lenguaje.

¿Qué opinas?

Patrones de Diseño

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20