1. Introducción

Los patrones de diseño facilitan la reutilización del conocimiento en un diseño. Un patrón es un fragmento identificable de conocimiento instructivo, que captura la estructura esencial e interna de una familia de soluciones con probado éxito sobre un problema recurrente dentro de un cierto contexto y fuerzas del software. En otras palabras, un patrón de diseño facilita reutilizar buenas soluciones.

2. Definición

En el enfoque orientada a objetos, un patrón de diseño es una realización probada (Diagrama de clases, objetos e interacción) que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de software orientado a objetos. Los patrones son un esqueleto básico que cada desarrollador luego adapta a sus necesidades y a las particularidades de su aplicación. Se describen fundamentalmente en forma textual, acompañados de diagramas y de seudo-código.

3. Historia de los patrones

El término patrón se utiliza inicialmente en el campo de la arquitectura, por Christopher Alexander, a finales de los 70s. Este conocimiento es trasportado al ámbito del desarrollo de software orientado por objetos y se aplica al diseño. De allí es extrapolado al desarrollo en general y a las demás etapas.

De acuerdo con Christopher Alexander en The Timeless Way of Building, 1.979, cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un problema y una solución. El patrón es al mismo tiempo una cosa que tiene su lugar en el mundo, y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es al mismo tiempo una cosa y un proceso; al mismo tiempo una descripción de una cosa que tiene vida y una descripción del proceso que la generó. Estos patrones en nuestras mentes son, más o menos, imágenes mentales de los

patrones en el mundo: son representaciones abstractas de las reglas morfológicas que definen los patrones en el mundo. Sin embargo, son realmente diferentes. Los patrones en el mundo solo existen. Pero esos mismos patrones en nuestras mentes son dinámicos. Tienen fuerza. Son generativos. Nos dicen qué hacer, cómo se pueden generar y, en ciertas circunstancias, que los debemos crear. Cada patrón es una regla que describe que debemos hacer para generar la entidad que los define.

Algunos libros que definen momentos históricos en la evolución de los patrones:

1. Alexander, Christopher. A Pattern Language: Towns, Buildings, Construction. 1977
2. Alexander, Christopher. The Timeless Way of Building. 1979
3. Gamma et al. Design Patterns: Elementos of Reusable Object-Oriented Software. 1994
4. Bushmann et al. Pattern-Oriented Software Architecture: A System of Patterns. 1996
5. Coplien y Schmidth. Pattern Languages of Program Design. 1995

4. Tipos de Patrones

Existen varios tipos de patrones, dependiendo del nivel de abstracción, del contexto particular en el cual aplican o de la etapa en proceso de desarrollo. Algunos de estos tipos son:

- De arquitectura: son esquemas fundamentales de organización de un sistema software. Especifican una serie de subsistemas y sus responsabilidades respectivas incluyendo tácticas y estrategias arquitectónicas para organizar las relaciones existentes entre ellos. Por ejemplo el estilo cliente-servidor desde un punto de vista de componentes y conectores.
- De diseño: son patrones de un nivel de abstracción alto pero a un nivel de granularidad más fino. Están por lo tanto más próximos a lo que serían las piezas concretas de la solución (código). Su uso no necesariamente se refleja en la estructura global del sistema. Por ejemplo el esquema observador-observado.

- Idioms: son patrones directamente involucrados en la codificación, por tanto son más simples y específicos al lenguaje. Por ejemplo, como iniciar un hilo(Thread) en Java.

5. Patrones de Diseño

Son patrones de un nivel de abstracción menor que los patrones de arquitectura y mayor que los idioms. Se caracterizan porque:

- Son soluciones concretas. Un catálogo de patrones es un conjunto de recetas de diseño. Aunque existen clasificaciones de patrones, cada uno es independiente del resto.
- Son soluciones técnicas. Dada una determinada situación, los patrones indican cómo resolverla mediante la OO. Hay patrones específicos para una plataforma determinada y otros de carácter más general.
- Se aplican en situaciones muy comunes. Los patrones de diseño proceden de la experiencia, y han demostrado su utilidad resolviendo problemas que aparecen frecuentemente en el diseño OO.
- Son soluciones simples. Indican cómo resolver un problema particular utilizando un pequeño número de clases relacionadas de forma determinada. No indican cómo diseñar un determinado sistema sino sólo aspectos puntuales del mismo.
- Facilitan la reutilización de las clases y del propio diseño. Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables. La propia estructura del patrón es reutilizada cada vez que se aplica.
- El uso de un determinado patrón no se refleja claramente en el código. A partir de la implementación es difícil determinar que patrón de diseño se ha usado.
- Referencias a this (self). Muchos patrones utilizan la delegación de operaciones y esto provoca el problema del this.
- Es difícil reutilizar la implementación de un patrón. Las clases del patrón son roles genéricos, pero en la implementación aparecen clases concretas.
- Los patrones suponen una sobrecarga de trabajo a la hora de implementar. Se usan más clases, es necesario delegar mensajes, etc.

Se pueden organizar los patrones según familias de patrones relacionados. La clasificación facilita la búsqueda del patrón más adecuado así como su comprensión. Gamma clasifica los patrones según dos criterios fundamentales: su propósito y su alcance. El propósito refleja lo que realiza el patrón. Los patrones pueden tener propósito de creación, estructural o de comportamiento. Los patrones con propósito de creación conllevan el proceso de creación de objetos. Los patrones estructurales tratan de la composición de clases u objetos. Los patrones de comportamiento describen las formas en que las clases u objetos interactúan o distribuyen responsabilidades.

El alcance indica si el patrón aplica principalmente a clases u objetos. Los patrones de clases tratan de relaciones entre clases y sus subclases. Estas relaciones se establecen a través de la relación de herencia, por consiguiente son estáticas y definidas en tiempo de compilación. Los patrones de objetos tratan de relaciones entre objetos que pueden ser cambiadas en tiempo de ejecución y son más dinámicas. Casi todos los patrones utilizan la herencia de alguna forma. Pero son los patrones de clases los que se focalizan en las relaciones de clase.

Los patrones con propósito de creación y alcance de clase difieren parte de la creación de objetos a subclases mientras que los patrones con propósito de creación y alcance de objeto difieren ésta a otros objetos. Los patrones estructurales y alcance de clase utilizan la herencia para componer clases, mientras que los patrones estructurales y alcance de objeto describen formas de ensamblado de objetos. Los patrones de comportamiento con alcance de clase utilizan herencia para describir algoritmos y flujos de control mientras que los patrones de comportamiento con alcance de objeto describen como un grupo de objetos cooperan para realizar una actividad que un objeto no puede realizar por sí solo.

6. Ejemplo de un patrón de diseño: Patrón Observador

6.1. Objetivo

Definir una dependencia uno-a-muchos entre objetos, de tal forma que cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados automáticamente. Se trata de desacoplar la clase de los objetos observadores del objeto observado, disminuyendo el acoplamiento entre las partes de la solución, creando las mínimas dependencias y evitando ciclos de actualización(espera activa). En defini-

tiva, normalmente, se usa el patrón Observador cuando un elemento quiere estar pendiente de otro, sin tener que estar preguntando de forma permanente si éste ha cambiado o no.

6.2. Motivación

Se necesita consistencia entre clases relacionadas, pero con independecia, es decir, un bajo acoplamiento.

6.3. Aplicabilidad

Este patrón aplica cuando una modificación en el estado de un objeto requiere cambios de otros, y no deseamos que se conozca el número de objetos que deben ser cambiados. También cuando queremos que un objeto sea capaz de notificar a otros objetos sin hacer ninguna suposición acerca de los objetos notificados y cuando una abstracción tiene dos aspectos diferentes, que dependen uno del otro; si encapsulamos estos aspectos en objetos separados permitiremos su variación y reutilización de modo independiente.

6.4. Estructura

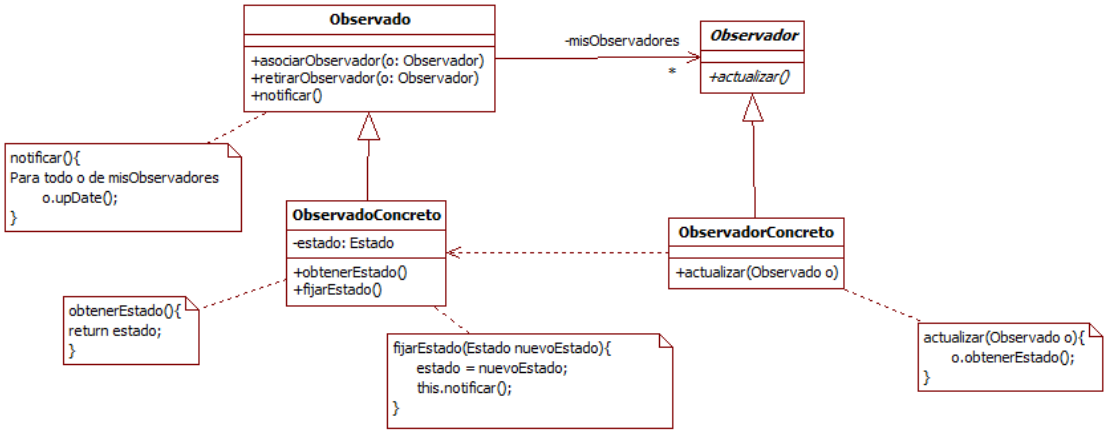

Figura 1: Patrón Observador

6.5. Participantes

Tendremos observados concretos cuyos cambios pueden resultar interesantes a otros y observadores a los que al menos les interesa estar pendientes de un elemento y en un momento dado, reaccionar ante sus notificaciones de cambio. Todos los observados tienen en común que un conjunto de objetos quieren estar pendientes de ellos. Cualquier elemento observado tiene que permitir indicar al observador:

1. Que el observador está interesado en sus cambios (asociar observador)
2. Que el observador ya no está interesado en los cambios (retirar observador)
3. El observado tiene que tener, además, un mecanismo de aviso a los observadores (actualizar).

Veamos cuales son los participantes:

- Observado: el observado proporciona una interfaz concreta para asociar y retirar observadores. El Observado conoce a todos sus observadores.
- Observador: define el método abstracto que usa el observado para notificar cambios en su estado (actualizar).
- ObservadoConcreto: mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.
- ObservadorConcreto: mantiene una referencia al observado concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio. Pueden no tener esta referencia y recibir una referencia al objeto observado a través del mismo método de actualización.

6.6. Colaboraciones

La colaboración más importante en este patrón es entre el observado y sus observadores, ya que en el momento en el que el observado sufre un cambio, este se lo notifica a sus observadores.

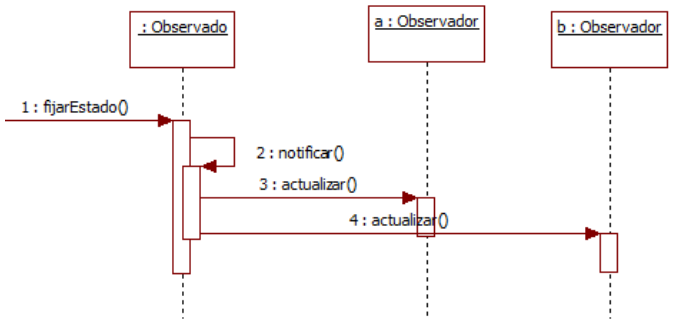

Figura 2: Colaboradores Patrón Observador

6.7. Consecuencias

Las consecuencias de aplicar este patrón pueden ser tanto beneficiosas, así como pueden perjudicar algunos aspectos. Por una parte abstrae el acoplamiento entre el observado y el observador, lo cual es beneficioso ya que conseguimos una mayor independencia y además el observado no necesita especificar los observadores afectados por un cambio. Por otro lado, con el uso de este patrón el observado desconoce las consecuencias de una actualización, lo cual, dependiendo del problema, puede afectarnos el comportamiento del observador o de todo el sistema (por ejemplo, el rendimiento).

6.8. Implementación

Algunos aspectos relacionados a la implementación del mecanismo de dependencia son discutidos en este apartado.

1. *Mapeo de los observados a sus observadores*: la forma más simple para que un observado conozca sus observadores y pueda notificar el cambio, es almacenar estas referencias explícitamente. Pero, este almacenamiento tiene mucho costo cuando son muchos los observados y pocos los observadores. Una solución es negociar espacio por tiempo usando una búsqueda asociativa (por ejemplo una tabla de hash) para mantener las referencias observado-observados. De ésta forma observados sin observadores no gastan en almacenamiento innecesario. Por otra parte, este enfoque incrementa el costo de acceder a los observadores.

2. *Observando más de un observado*: es posible que en algunas situaciones un

observador dependa de más de un observado. Por ejemplo una hoja electrónica puede depender de más de una fuente de datos. En este caso es necesario extender la interface *actualizar()* para permitir que el observador pueda saber cuál es el objeto que está enviando la notificación. El Observado puede pasarse a sí mismo (this, self) como un parámetro en la operación *actualizar()* para que el observador pueda conocer el objeto observado. En este caso la interface quedaría así *actualizar(Observado obs)*.

3. *¿Quién dispara la actualización?* el observado y sus observadores usan la notificación para mantener la consistencia. Sin embargo, ¿cuál es el objeto que realmente llama a *notificar()* para disparar la actualización?:

   a) En las operaciones de cambio de estados en el observado (setVariable), donde el observado llama el método notificar después del cambio de estado. La ventaja de éste enfoque es que los clientes no tienen que recordar el llamado a la operación notificar en el observado. La desventaja es que algunas operaciones consecutivas causarán algunas actualizaciones consecutivas, lo cual puede resultar ineficiente.

   b) Hacer a los clientes responsables de llamar el método notificar en todo momento. La ventaja aquí es que el cliente puede esperar a disparar la actualización hasta después de que una serie de cambios de estados se hagan, evitando actualizaciones intermedias. La desventaja es que los clientes tienen la responsabilidad adicional de disparar la actualización. El error más común en éste caso es que los clientes pueden olvidar el llamado a la notificación.

4. *Referencias colgantes a observados eliminados*: el borrado de un observado no debería producir referencias nulas en sus observadores. Una forma de evitar referencias colgantes es hacer que el observado notifique a sus observadores cuando él es eliminado, para que el observador pueda poner en nulo su referencia a éste. En general, el sólo borrado de los observadores no es una opción , porque otros objetos pueden estar referenciándolos, o éstos podrían estar observado otros observados.

5. *Estar seguro que el estado del observado es auto-consistente antes de la notificación*: es importante estar seguro que el estado del observado es auto-consistente antes del llamado a notificar debido a que los observadores consultan el observado por su estado actual con el fin de actualizar su propio estado. La regla de auto-consistencia es fácil de violar sin intensión cuando las operaciones de las

subclases del observado llaman operaciones heredadas. Por ejemplo, la notificación de la siguiente secuencia de código es disparada cuando el es observado está en un estado inconsistente:

```
class Observado{
  void operation(int newValue){
    super.operation(newValue); // dispara la notificación
    instanceValue += newValue;
    notificar();
  }
}
```

Se puede evitar el este error enviando notificaciones desde métodos template (patrón template) en una clase abstracta Observado. Definir una operación primitiva en la subclase para sobreescribir y hacer de notificar la última operación en el método template, lo cual asegura que el objeto estará autoconsistente cuando la subclase sobre-escribe las operaciones del Observado.

```
class Texto extends Observado{
  void cortar(RangoTexto r){
    reemplazarRango(r); // será sobreescrita en las subclases
    notificar();
  }
}
```

De ésta forma, es una buena idea documentar todas las operaciones del observador que disparen notificaciones.

6. *Evitar protocolos de actualización específicos al observador: los modelos push(empujar) y pull(jalar)*: las implementaciones del patrón Observador frecuentemente tienen que transmitir información adicional acerca del cambio. El observado pasa ésta información como un argumento a actualizar. La cantidad de información puede variar ampliamente. En un extremo, el *modelo push*, el observado envía a los observadores información detallada acerca del cambio, así éstos la requieran o no. En el otro extremo, el *modelo pull*, el observado no envía información de cambio sino la más mínima notificación, en éste caso los observadores consultan posteriormente los detalles en forma explícita. El *modelo pull* enfatiza en el desconocimiento que los observados tienen de sus observadores, mientras el modelo *modelo push* supone que los observados saben de las necesidades de los observadores y por tanto es menos reutilizable, puesto que los observados hacen suposiciones de los observadores que no siempre son necesariamente ciertas.

Por otro laso el *modelo pull* puede resultar ineficiente, debido a que las clases Observadoras deben averiguar qué cambió sin la ayuda del observado.

7. *Especificar modificaciones de interés explícitamente*: se puede mejorar la eficiencia de la actualización extendiendo la interface de registro del observable para permitir el registro de observadores solamente para eventos específicos de interés. Cuando éstos eventos ocurren el observado solamente notifica sólo a aquellos observadores que tienen un interés registrado en ese evento. Una forma de soportar esto usa la noción de *Evento de Interés* para los objetos observados. Para registrar el interés en ciertos eventos, los observadores son adjuntados a sus observados usando la interface:

```
class Observado{

asociarObservador(Observador obs, EventoInteres e){
  // codigo de asociación
}

}
```

Donde EventoInteres especifica el evento de interés. En el momento de la notificación el observado suministra el evento de interés cambiado como un parámetro en la operación de actualización. Por ejemplo:

```
class Observador{

actualizar(Observado obs, EventoInteres e){
  // codigo de actualización
}

}
```

8. *Encapsular semántica de actualización compleja*: cuando la relación de dependencia entre los observados y los observadores es particularmente compleja, se requiere de un objeto que mantenga esas relaciones. Llamaremos a ese objeto un *Controlador de Cambio*. Su propósito es minimizar el trabajo requerido para hacer que los observadores reflejen los cambios de sus observados. Por ejemplo, en el caso en que una operación involucre cambios de algunos observados interdependientes, debemos asegurarnos que los observadores son notificados solamente después de que *todos* los objetos observados han sido modificados

a fin de evitar notificar a los observadores más de una vez. El *Controlador de Cambio* tiene tres responsabilidades:

a) Este asocia un observado a sus observadores y provee una interface para mantener esta relación. Esto elimina la necesidad de que los observados mantengan referencias a sus observadores y viceversa.

b) Este define una estrategia particular de actualización.

c) Este actualiza todos los observadores dependientes de la solicitud de un observado.

9. *Combinando las clases Observado y Observable*: librerías de clases escritas en lenguajes que no tienen herencia múltiple(como Java y Smalltalk), generalmente definen Observados y Observables en clases separadas y combinan sus interfaces en una misma clase, esto permite definir objetos que son observados y observable al mismo tiempo. En Smalltalk por ejemplo la clase Object implementa la especificación de observado y observador, de tal forma que todas las clases hereden este comportamiento.

6.9. Código Fuente

La interface Observador puede ser definida por una clase abstracta o una interface:

```
public interface Observador {
     public void actualizar(Observable o);
}
```

La clase Observada es una clase concreta que tiene implementado el servicio de notificación:

```
public class Observado{
    private LinkedList<Observador> observadores;

    public void agregarObservador(Observador obs){
        if(observadores == null)
            observadores = new LinkedList<Observador>();
        observadores.add(obs);
    }
```

```
    public void notificar(){
        for(Iterator<Observador> e =
                observadores.iterator(); e.hasNext(); ){
                        e.next().actualizar(this);
        }
    }
}
```

Consideremos ahora un Reloj como un Observado concreto para almacenar y manipular la hora del día. Éste notifica a sus observadores cada vez de cambia. Durante la operación avanzar en la clase Reloj, el reloj cambia de estado y por tanto llama a la operación *notificar*.

```
public class Reloj extends Observable {

    private Manecilla menor;

    public Reloj(Manecilla menor){
        this.menor = menor;

    }
    public void correr(){
         while(true)
        {
            menor.incrementar();
            this.notificar();
        }
    }

    public String getHora(){return menor.getValue();}
}
```

Ahora podemos definir la clase VisualizadorReloj la cuál se encarga de desplegar el tiempo. Esta hereda la funcionalidad desde un JFrame de la API de J2SE e implementa la interface Observador. Antes de que la operación *actualizar* dibuje el reloj, se chequea que el observado notificado sea el Observado asociado.

```
public class VisualizadorReloj extends JFrame
       implements Observador {
    private Reloj relojito = null;
     @Override
    public void actualizar(Observable o){
        if(relojito == o)
        mostrar((Reloj) o);
    }

    public void clickInicio(){
     relojito.correr();
          }

    public void mostrar(){
        etiquetaHora.setText(relojito.getHora());
        this.repaint();
    }
 }
```

También podemos definir una clase Alarma, la cual se puede activar a cierta hora específica. Ésta también implementa la interface Observador, en este caso se verifica que la hora actual corresponde con la hora de la alarma, de ser así activa un sonido repetitivo.

```
public class Alarma {
    private Reloj relojito = null;
    private String horaAlarma;
     @Override
    public void actualizar(Observable o){
        if((relojito == o) && (relojito.getHora()==horaAlarma))
          this.activar();
    }

     public void activar(){
        // Código de activar alarma
    }
 }
```

Ahora crearemos un programa principal:

```
public class AplicativoReloj{
  VisualizadorReloj vistaReloj;
        Manecilla hora;
        Manecilla min;
        Manecilla seg;
        Manecilla menor;
        Reloj relojito;

        public AplicativoReloj(){}
                hora = new Manecilla(0,24,23,null);
                min = new Manecilla(0,60,59,hora);
                seg = new Manecilla(0,60,50,min);
                relojito = new Reloj(seg);
                vistaReloj = new VisualizadorReloj();
                relojito.agregarObservador(vistaReloj);
                alarmita = new Alarma();
                relojito.agregarObservador(alarmita);
        }



  public static void main(String args[]){
    AplicativoReloj app = new AplicativoReloj();
    app.setVisible(true);

  }
}
```

Cuando el programa inicia, crea el reloj y sus partes, así como la configuración de sus observadores. Cuando el usuario hace click en el botón iniciar, el reloj empieza a correr. A cada cambio el VisualizadorReloj se actualiza, mientras la Alarma sólo evalúa si es la hora de activarse, lo cual hará cuando su hora de activación coincida con la hora actual.

6.10. Usos Conocidos

El uso más concido del patrón es la solución del framework de interface de usuario de Smalltalk Modelo-Vista-Controlador, donde el Modelo es el Observado y la Vista es el observador, el controlador es un intermediario que facilita la observación de los eventos registrados.

6.11. Patrones Relacionados

Patrón Mediador, cuando la semántica de actualización es compleja, por ejemplo un Manejador de Eventos, donde actua como un mediador entre observadores y observados.

El patrón Singleton, el Manejador de Eventos podría ser una sola instancia accesible como una variable global.

7. Descripción de un Patrón de Diseño

Dependiendo del autor, del nivel de abstracción y de la publicación misma se han presentado varios formatos para encapsular la información de un patrón. Los puntos más significativos que debe contener un patrón son:

1. Nombre: Significativo y corto, fácil de recordar y asociar a la información que sigue
2. Problema: Un enunciado que describe las metas y objetivos buscados en el contexto
3. Contexto: Define las precondiciones en las cuales ocurren el problema y su solución
4. Fuerzas: Descripción de las fuerzas y restricciones relevantes en el problema y como interactúan o entran en conflicto
5. Solución: Las relaciones estáticas y reglas dinámicas que describen cómo solucionar el problema
6. Ejemplos: Uno o más ejemplos que ilustren el contexto, el problema y su solución

7. Contexto Resultante: El estado en el cual queda el sistema después de aplicar el patrón y las consecuencias de hacerlo

8. Rationale: Una explicación justificada de los pasos o reglas en el patrón

9. Relaciones estáticas y dinámicas del patrón con otros patrones

10. Usos conocidos: Describe ocurrencias del patrón conocidas y su aplicación dentro de los sistemas existentes

La propuesta de Gamma (1995), por ejemplo, propone que los elementos esenciales de un patrón son los siguientes:

1. Un nombre del patrón. Es una forma abreviada que pueda darnos una idea del problema al que se aplica, sus soluciones y consecuencias. Al asignar un nombre, estamos facilitando la tarea de diseño puesto que nos comunicamos a un mayor nivel de abstracción. Es bastante difícil encontrar nombres adecuados que sirvan a este propósito.

2. El problema describe cuando aplicar el patrón. Aquí se explica el problema y su contexto. Un añadido útil es el de las condiciones de aplicabilidad del patrón.

3. La solución describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. Se insiste mucho en que esta solución es como una plantilla que provee una descripción abstracta de un problema de diseño y cómo una disposición general de elementos (en este caso clases y objetos) puede resolverlo.

4. Las consecuencias son los resultados y compromisos de aplicar el patrón.

Estos son los elementos esenciales, pero cuando se trata de realizar una descripción concreta de un patrón, la plantilla propuesta estará compuesta por una serie de secciones que permiten una estructura más detallada que la ofrecida por la enumeración de los elementos esenciales. Siguiendo a Gamma, encontramos la siguiente lista y descripción de secciones dentro de la plantilla que describe cada patrón. Este formato estructurado es útil puesto que permite la separación semántica de lo que podría haber sido un texto completo y permite además su almacenamiento en una base de datos para su posterior acceso si deseamos aumentar su reutilización.

1) Nombre del Patrón y Clasificación, 2) Intención, 3) También conocido como (Sinónimo), 4) Motivación, 5) Aplicabilidad, 6) Estructura, 7) Participantes, 8) Colaboraciones, 9) Consecuencias, 10) Implementación, 11) Ejemplo de código, 12) Usos conocidos y Patrones relacionados.

8. Conclusiones

Los patrones de diseño son aporte significativo a la ingeniería de software porque:

1. Establecen un vocabulario común: el nombre del patrón y su clasificación son mecanismos de almacenamiento de experiencia y conocimiento, tanto en repositorios como en la mente de los mismos diseñadores.

2. Promueven un buen diseño OO: dado que aplica esquemas extraídos de diseños de expertos, su aplicación es confiable por el uso previamente probado de las soluciones que ofrece.

3. Sirven para preparar nuevos diseñadores: los patrones son un punto de partida en dónde se reutiliza el conocimiento, así que son una herramienta clave en la capacitación de las organizaciones en aspectos relacionados con el diseño de software.

4. Estandarizan la forma en que los diseños son desarrollados: dado que definen esquemas de implementación, si se usan sus nombres y estructuras, le darán una homogeneidad a las implementaciones realizadas, lo cual es importante al momento de hacer mantenciones.

5. Enseñan la Orientación a Objetos: la orientación a objetos es un arma poderosa para hacer diseños reutilizables y flexibles. Es con el uso de patrones dónde aflora toda esa capacidad.

6. Ayudan a hacer refactorizaciones: diseños legados o defectuosos pueden ser mejorados via la refactorización, en ese proceso varios patrones pueden ser utilizados para facilitar el mejoramiento del producto.