Patrones de Diseño, Refactorización y Antipatrones. Ventajas y Desventajas de su Utilización en el Software Orientado a Objetos

Gustavo Damián Campo[*]

guscampo@yahoo.com.ar

Resumen

El presente artículo busca introducir al lector en los conceptos de Patrón de Diseño, Refactorización y Antipatrón, y cuáles son las ventajas y desventajas inherentes a su aplicación en el desarrollo de software orientado a objetos. Este no es un artículo sobre orientación a objetos, por lo que no se tratan temas como herencia, polimorfismo y encapsulamiento, entre otros.

Palabras Claves: Patrones de Diseño, Patrones, Refactorización, Antipatrones

1. Introducción

A lo largo de los años la ingeniería de software fue introduciendo nuevas técnicas en pos de obtener software de mejor calidad, tales como las técnicas estructuradas, los lenguajes de cuarta generación, las herramientas CASE, la orientación a objetos y nuevos modelos de calidad. Investigadores académicos y profesionales desarrollaron distintos enfoques innovadores para la construcción de software, desde el uso de nuevas tecnologías hasta la aplicación de nuevos procesos. Incluso con todas estas ideas, el éxito de un producto software nunca estuvo garantizado.

[*] El autor es Ingeniero en Informática por la Universidad Católica de Salta, Subsede Buenos Aires. En el ámbito laboral, se dedica al desarrollo de software. El presenta artículo es una síntesis de su trabajo de tesis.

Casi todos los sistemas entregados son, en mayor o menor medida, sistemas diseñados para un determinado propósito y con poca flexibilidad, lo que se conoce como *stovepipe systems*, sistemas que no pueden adaptarse al cambio. Este factor debe considerarse siempre ya que el grado de flexibilidad de un producto software está directamente relacionado con la calidad del mismo.

Si bien la orientación a objetos mejora la productividad, adaptabilidad y reutilización del software, los sistemas orientados a objetos con muchas líneas de código son difíciles de mantener y extender con nueva funcionalidad. Esto no quiere decir que conceptos tales como herencia, encapsulamiento o polimorfismo no sean adecuados, sino que por sí solos no son suficientes. Estas técnicas deben combinarse con otras de mayor nivel para poder obtener software flexible, pero para poder hacerlo hace falta algo: experiencia. Es esta experiencia la que hace que los expertos resuelvan nuevos problemas mediante la aplicación de resoluciones que funcionaron con anterioridad para problemas similares.

A la hora de recolectar experiencia, se pueden armar dos grandes grupos: las buenas y las malas:

- Los Patrones de Diseño se corresponden con las buenas experiencias, ya que reúnen resoluciones a problemas recurrentes encontrados en el diseño de software orientado a objetos.

- Los Antipatrones, por su parte, se corresponden a las malas experiencias ya que reúnen soluciones que han producido efectos negativos. Los Antipatrones proveen dos soluciones: la problemática (aquélla con el impacto negativo), y la refactorizada (aquélla que transforma la situación negativa en una más saludable).

Como se dijo anteriormente, la utilización de herencia, encapsulamiento y polimorfismo debe combinarse con técnicas de mayor nivel. Estas técnicas de mayor nivel son: los Patrones de Diseño, para obtener software flexible; los Antipatrones para evitar cometer errores recurrentes; y la Refactorización para asegurar reorganizaciones ordenadas de código fuente para mejorar la mantenibilidad, o salir de algún Antipatrón.

El inicio de los Patrones de Diseño suele atribuirse a un profesor de arquitectura de la Universidad de Berkeley llamado Christopher Alexander. *A Pattern Language* (1977) y *The Timeless Way of Building*

(1979) presentaron la visión del autor sobre problemas recurrentes que existían en la arquitectura de pueblos, ciudades y en cualquier tipo de construcción. Estos problemas y soluciones fueron descriptos por Alexander como “patrón”.

El trabajo de Alexander despertó el interés en el tema por parte de la comunidad de programadores de software orientado a objetos y dentro de la siguiente década comenzaron los desarrollos de patrones para el diseño de software.

Con el correr de los años, los patrones fueron ganando aceptación y se han ido extendiendo a diversas áreas de la construcción de software.

Patrones de Diseño y Antipatrones brindan una forma efectiva de compartir experiencia. Sea cual sea el lenguaje de programación utilizado, las lecciones aprendidas pueden recolectarse con el fin de ser compartidas con otros desarrolladores. A largo plazo, esto podría mejorar la industria de desarrollo de software.

2. 2 Patrones de diseño

Un Patrón de Diseño (*design pattern*) es una solución repetible a un problema recurrente en el diseño de software. Esta solución no es un diseño terminado que puede traducirse directamente a código, sino más bien una descripción sobre cómo resolver el problema, la cual puede ser utilizada en diversas situaciones. Los patrones de diseño reflejan todo el rediseño y remodificación que los desarrolladores han ido haciendo a medida que intentaban conseguir mayor reutilización y flexibilidad en su software.

Los patrones documentan y explican problemas de diseño, y luego discuten una buena solución a dicho problema. Con el tiempo, los patrones comienzan a incorporarse al conocimiento y experiencia colectiva de la industria del software, lo que demuestra que el origen de los mismos radica en la práctica misma más que en la teoría.

En (Gamma et al, 1995) se cita la definición de Christopher Alexander sobre patrones: “cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces”. Si bien ésta definición es sobre

patrones de ciudades y edificios la idea es aplicable a la industria del software: encontrar una solución a un problema dentro de un contexto.

Un patrón de diseño nomina, abstrae e identifica los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizable. El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se centra en un problema concreto, describiendo cuándo aplicarlo y si tiene sentido hacerlo teniendo en cuenta otras restricciones de diseño, así como las consecuencias, ventajas e inconvenientes de su uso.

En los últimos años los patrones han ido ganando aceptación, y se fueron extendiendo a otras áreas dentro del desarrollo y mantenimiento de software. Su utilización, si bien todavía le queda mucho camino por recorrer, comienza a tener suficiente madurez.

Los patrones de diseño proveen una forma efectiva para compartir experiencia con la comunidad de programadores de software orientado a objetos.

2.1. Componentes

Para que una solución sea considerada un patrón debe poseer ciertos componentes fundamentales:

- Nombre del patrón. Permite describir en pocas palabras un problema de diseño junto con sus soluciones y consecuencias.

- Problema (o fuerzas no balanceadas). Indica cuándo aplicar el patrón. En algunas oportunidades el problema incluye una serie de condiciones que deben darse para aplicar el patrón. Muestra la verdadera esencia del problema. Este enunciado se completa con un conjunto de fuerzas, término que se utiliza para indicar cualquier aspecto del problema que deba ser considerado a la hora de resolverlo, entre ellos:
  - Requerimientos a cumplir por la solución.
  - Restricciones a considerar.
  - Propiedades deseables que la solución debe tener.

Son las fuerzas las que ayudan a entender el problema dado que lo exponen desde distintos puntos de vista.

- Solución. No describe una solución o implementación en concreto, sino que un patrón es más bien como una plantilla que puede aplicarse en diversas situaciones diferentes. El patrón brinda una descripción abstracta de un problema de diseño y cómo lo resuelve una determinada disposición de objetos. Que la solución sea aplicable a diversas situaciones denota el carácter "recurrente" de los patrones.

- Consecuencias. Resultados en términos de ventajas e inconvenientes.

2.2. Tipos

Según el nivel de abstracción los patrones de diseño pueden clasificarse de la siguiente forma:

- Patrones arquitectónicos. Centrados en la arquitectura del sistema. Definen una estructura fundamental sobre la organización del sistema. Proveen un conjunto predefinido de subsistemas, cuáles son sus responsabilidades y como se interrelacionan.

- Patrones de diseño. Esquemas para refinar los subsistemas o componentes de un sistema de software, o sus relaciones. Describen una estructura recurrente y común de componentes comunicantes que resuelven un problema de diseño dentro de un contexto. Ejemplo: el patrón *singleton* asegura que exista sólo una instancia de una determinada clase.

- Patrones de codificación o modismos (*idioms*). Patrones que ayudan a implementar aspectos particulares del diseño en un lenguaje de programación específico. Ejemplo: en Java implementar una interface en una clase anónima.

Los patrones arquitectónicos podrían considerarse estrategias de alto nivel que abarcan componentes a gran escala, propiedades y mecanismos del sistema. Tienen implicancias muy amplias que afectan tanto a la estructura como a la organización del sistema. Los patrones de diseño son tácticas de medio nivel para profundizar en la estructura y comportamiento de ciertos componentes y sus relaciones. Los

patrones de diseño no influencian la estructura del sistema sino que definen micro-arquitecturas para los subsistemas y componentes. Por último, los modismos son técnicas específicas del paradigma y lenguaje de programación que complementan detalles de bajo nivel (internos o externos) de la estructura de un componente.

2.3. Preconceptos equivocados

Es importante aclarar algunas cuestiones que pueden llevar a confusión sobre qué es un patrón y lo que proveen. En (Vlissides, 1997) se listan los principales conceptos equivocados sobre patrones de diseño, a continuación se indican algunos de ellos.

2.3.1. Conceptos equivocados sobre qué son los patrones de diseño

- "Un patrón es una solución a un problema en un contexto". Si bien esta es una definición de Christopher Alexander, hay algunos factores adicionales a considerar:
  - Recurrencia: lo que hace relevante a la solución para situaciones diferentes que la inmediata.
  - Enseñanza: debe proveer el entendimiento suficiente como para personalizar la solución ofrecida al nuevo problema.
  - Nombre: debe proveer un nombre por el cual referirse al patrón.

  Cualquier definición que indique las partes constituyentes de un patrón debe expresar recurrencia, enseñanza, y nombre adicionalmente al problema, solución y contexto.

- "Los patrones son simplemente jerga, reglas y trucos de programación". Comparado otras áreas de la informática, los patrones introducen relativamente pocos términos nuevos. Un buen patrón es intrínsicamente accesible a su audiencia. Puede utilizar la jerga del dominio en cuestión, pero es escasa la necesidad de utilizar terminología específica de patrones. Tampoco son reglas que se pueden aplicar ciegamente, ya que la componente de enseñanza debería impedir esta tendencia.

- "Los patrones necesitan herramientas o soporte metodológico para ser efectivos". Uno de los grandes beneficios de los patrones es que pueden ser aplicados directamente, sin ningún tipo de soporte.

Por su parte (Piatini y Garcia, 2003) sostienen esto mismo al establecer que los patrones no son:

- Invenciones, teorías o ideas no probadas
- Soluciones que sólo han funcionado una vez
- Principios abstractos o heurísticos
- Aplicaciones universales para cualquier contexto.

2.3.2. Conceptos equivocados sobre qué brindan, o garantizan, los patrones de diseño

- "Los patrones garantizan software reutilizable, mayor productividad…" Los patrones por si solos no garantizan nada. Los patrones no excluyen a las personas del proceso creativo. Simplemente proveen esperanza de mejora a una persona posiblemente inexperta, recientemente iniciada en el tema, pero capaz y creativa. Los patrones no son más que otra herramienta del desarrollador.

- "Los patrones generan arquitecturas completas". La capacidad de generación de un patrón se encuentra en la explicación sobre las fuerzas y su solución, o en la discusión de las consecuencias del patrón. Estos conocimientos son especialmente útiles a la hora de definir y refinar una arquitectura. Pero creer que los patrones por si solos generan arquitecturas o cualquier otra cosa es erróneo. Los patrones no generan nada, sólo la gente lo hace.

- "Los patrones son para diseño o implementación (orientado a objetos)". Los patrones no son nada si no capturan experiencia. La naturaleza de esa experiencia es dejada al autor del patrón. Existe mucha experiencia que vale la pena capturar en el diseño orientado a objetos, pero también la hay otras áreas: diseño no orientado a objetos, mantenimiento, análisis, pruebas, documentación, estructura organizacional, etc.

  Un formato de patrones no alcanza para todas las disciplinas, pero lo que sí lo hace es el concepto de patrón como un vehículo para capturar y transmitir experiencia, sin importar la disciplina. "Un patrón es una solución a un problema en un contexto

2.4. Catálogo

Los patrones de diseño refinan subsistemas o componentes de software orientado a objetos. Estos describen una estructura genérica

sobre el problema a resolver, de manera tal de poder contextualizarla con el problema a resolver. (Fowler, 2003) sostiene que “están a medio cocer” ya que siempre se los debe contextualizar al entorno en el que serán aplicados. De hecho, una de las mejores formas de aprender sobre ellos es implementarlos uno mismo.

El principal catálogo de patrones de diseño es (Gamma et al, 1995), el cual consta de 23 patrones de diseño agrupados en tres grupos: Patrones Creacionales, Patrones Estructurales y Patrones de Comportamiento. En las siguientes secciones se lista cada uno de ellos junto con un breve ejemplo conceptual que ilustra la implementación básica del patrón.

2.4.1 Patrones Creacionales

Los patrones creacionales abstraen el proceso de instanciación de objetos, ayudando a que el sistema sea independiente de cómo se crean, componen y representan sus objetos. Estos patrones encapsulan el conocimiento sobre las clases concretas que utiliza el sistema.

En otras palabras, estos patrones brindan soporte a una de las tareas mas comunes dentro de la programación orientada a objetos: la instanciación. Estos patrones brindan las siguientes características:

- Instanciación genérica: permite que los objetos sean creados dentro del sistema sin especificar clases concretas en el código.
- Simplicidad: algunos patrones facilitan la creación de objetos, evitando que el cliente deba tener código complejo sobre como instanciar un determinado objeto.
- Restricciones creacionales: algunos patrones ayudan a establecer restricciones sobre la creación de objetos, tales como qué objeto crear, cuándo, cómo, etc.

Por lo general, son alternativas de diseño bajo estrategias de herencia o delegación que encapsulan el mecanismo de creación, independizando los tipos de objetos “producto” que se manejan.

Los patrones creacionales son los siguientes:

1. Singleton: asegura que una determinada clase sea instanciada una y sólo una vez, proporcionando un único punto de acceso global a ella.

2. Abstract Factory: provee una interfaz para crear familias de objetos producto relacionados o que dependen entre si, sin especificar sus clases concretas.

3. Factory Method: define una interfaz para crear un objeto delegando la decisión de qué clase crear en las subclases. Este enfoque también puede ser llamado constructor “virtual”.

4. Builder: separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones. Simplifica la construcción de objetos con estructura interna compleja y permite la construcción de objetos paso a paso.

   Ejemplo: este patrón se encuentra en los restaurantes de comidas rápidas que preparan menús infantiles. Generalmente estos menús están formados de un plato principal, un acompañamiento, una bebida y un juguete. Si bien el contenido del menú puede variar, el proceso de construcción es siempre el mismo: el cajero indica a los empleados los pasos a seguir. Estos pasos son: preparar un plato principal, preparar un acompañamiento, incluir un juguete y guardarlos en una bolsa. La bebida se sirve en un vaso y queda fuera de la bolsa.

5. Prototype: facilita la creación dinámica de objetos mediante la definición de clases cuyos objetos pueden crear duplicados de si mismos. Estos objetos son llamados prototipos.

   Ejemplo: un escenario frecuente es contar con GUIs (Interfaz Gráfica de Usuario) que cuenten con un gran número de controles similares, los cuales deben ser inicializados a un determinado estado común para mantener consistencia. El proceso de inicialización se repite varias veces por cada control de manera que las líneas de código se incrementan. Con el fin de optimizar estas partes del código se puede contar con un objeto inicializado en un determinado estado estándar y luego obtener clones de él ya inicializados.

2.4.2 Patrones estructurales

Se encargan de cómo se combinan clases y objetos parar formar estructuras más grandes. Los patrones estructurales de clases utilizan la herencia para componer interfaces o implementaciones. En lugar de combinar interfaces o implementaciones, los patrones estructurales de

objetos describen formas de componer objetos para obtener nuevas funcionalidades. La flexibilidad añadida mediante la composición de objetos viene dada por la capacidad de cambiar la composición en tiempo de ejecución, que es imposible con la composición de clases. Ejemplos típicos son cómo comunicar dos clases incompatibles o cómo añadir funcionalidad a objetos.

Los patrones estructurales son los siguientes:

6. Adapter: oficia de intermediario entre dos clases cuyas interfaces son incompatibles de manera tal que puedan ser utilizadas en conjunto.

   Ejemplo: en la vida cotidiana se ven ejemplos de este patrón, quizá el mas común de ellos sea el de los adaptadores de enchufes el cual permitiría utilizar un enchufe de dos patas planas adaptándolo a un toma corriente de dos patas redondas.

7. Bridge: disocia un componente complejo en dos jerarquías de clases: una abstracción funcional y la implementación interna, para que ambas puedan variar independientemente.

   Ejemplo: los electrodomésticos y sus interruptores de encendido pueden ser considerados como ejemplos de este patrón donde el interruptor de encendido es considerado la abstracción y el electrodoméstico en si la implementación. El interruptor podría ser un simple interruptor de encendido/apagado, un regulador de velocidades u alguna otra opción, mientras que el electrodoméstico puede ser una lámpara, un ventilador de techo, etc.

8. Composite: compone objetos en estructuras de árboles para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los complejos.

   Ejemplo: en una gráfica de Gantt existen tareas simples (con una actividad) y compuestas (que contienen varias tareas). Modelar estos dos tipos de tareas en una jerarquía de clases donde ambas son subclases de una clase que cuente con un método “calculáTiempoUtilizado” permitiría tratar de forma uniforme a tareas simples y compuestas para calcular el tiempo utilizado por cada una de ellas. Concretamente una tarea simple informa el tiempo dedicado a ella, mientras que una compuesta lo hace sumando los tiempos insumidos de cada una de las tareas que contiene.

9. Decorator: agrega o limita responsabilidades adicionales a un objeto de forma dinámica, proporcionando una alternativa flexible a la herencia para extender funcionalidad.

   Ejemplo: si bien es cierto que se pueden colgar pinturas, cuadros y fotos en las paredes sin marcos, éstos suelen ser utilizados a menudo y son ellos los que se cuelgan en la pared en lugar de su contenido (pinturas, cuadros, etc.). Al momento de colgarse los cuadros junto con su marco pueden formar un solo “componente visual”

10. Facade: proporciona una interfaz simplificada para un conjunto de interfaces de subsistemas. Define una interfaz de alto nivel que hace que un subsistema sea más fácil de usar.

    Ejemplo: En un sistema de compras los clientes contactan a un responsable de ventas que actúa como Facade al momento de realizar un pedido. Este representante de ventas actúa como Facade proveyendo una interface con los departamentos (subsitemas) de pedidos, facturación y envíos.

11. Flyweight: permite el uso de un gran número de objetos de grano fino de forma eficiente mediante compartimiento.

    Ejemplo: La red telefónica pública conmutada es un ejemplo de este patrón ya que hay diversos componentes, como por ejemplo nodos de conmutación, que se deben compartir entre los distintos usuarios. Los usuarios no conocen cuántos componentes de cada tipo hay disponibles al momento de realizar la llamada. Lo único por lo que se preocupan los usuarios es por obtener tono para marcar, poder discar y efectuar la llamada.

12. Proxy: Provee un sustituto o representante de un objeto para controlar el acceso a éste. Este patrón posee las siguientes variantes:
    - Proxy remoto: se encarga de representar un objeto remoto como si estuviese localmente.
    - Proxy virtual: se encarga de crear objetos de gran tamaño bajo demanda.
    - Proxy de protección: se encarga de controlar el acceso al objeto representado.

2.4.3. Patrones de comportamiento

Tienen que ver con algoritmos y asignación de responsabilidades. Estos patrones se focalizan en el flujo de control dentro de un sistema. Ciertas formas de organizar los controles dentro del sistema pueden llevar a grandes beneficios en cuanto a mantenibilidad y eficiencia. Algunos ejemplos de estos patrones incluyen la definición de abstracciones de algoritmos, las colaboraciones entre objetos para realizar tareas complejas reduciendo las dependencias o asociar comportamiento a objetos e invocar su ejecución. Los patrones de comportamiento basados en clases utilizan la herencia para distribuir el comportamiento entre clases, ellos son: Template Method e Interpreter. Mientras que los basados en objetos utilizan la composición.

Los patrones de comportamiento son los siguientes:

13. Chain of responsibility: establece una cadena de mensajes dentro del sistema de manera tal que dicho mensaje sea manejado en el mismo nivel donde fue emitido, o redirigido a un objeto capaz de manejarlo. Evita acoplar el emisor del mensaje con un receptor, dando a más de un objeto la posibilidad de responder al mensaje.

14. Command: representa una solicitud con un objeto, de manera tal de poder parametrizar a los clientes con distintas solicitudes, encolarlas o llevar un registro de las mismas, y poder deshacer las operaciones. Estas solicitudes, al ser representadas como un objeto también pueden pasarse como parámetro o devolverse como resultados.

15. Interpreter: en un contexto donde se repite una determinada clase de problemas y el dominio es bien conocido, se pueden caracterizar estos problemas como un lenguaje y, a su vez, estos problemas pueden ser tratados por un “motor” de interpretación. Este patrón busca definir un intérprete para dicho lenguaje, para el cual define una gramática y un intérprete de la misma para poder resolver los problemas.

    Ejemplo: distintos motores de bases de datos (Oracle, SQL Server, Sybase, DB2, etc.) utilizan distintos códigos de error para indicar fallas (errores de clave duplicada, violación de restricciones de integridad referencial, longitud de datos, etc.). La utilización de éste patrón permitiría definir un intérprete de errores para cada motor de base de datos con el cual se determinaría la falla y tomarían las acciones pertinentes en función de la misma. El sistema debe

configurarse para utilizar el interprete adecuado según el motor de base de datos.

16. Iterator: provee un modo de acceder secuencialmente a los elementos de un objeto agregado (una colección) sin exponer su representación interna. El iterador está altamente acoplado al objeto agregado.

    Ejemplo: Los árboles-B pueden recorrerse de tres formas distintas: pre-orden, en-orden y post-orden. La aplicación de este patrón permitiría definir un iterador para cada tipo de recorrido, pudiendo ser utilizados para recorrer el árbol sin exponer su contenido.

17. Mediator: simplifica la comunicación entre objetos dentro del sistema mediante la introducción de un objeto mediador que administra la distribución de mensajes entre objetos. Promueve bajo acoplamiento al evitar que los objetos se referencien unos a otros explícitamente, permitiendo variar la interacción entre ellos independientemente.

    Ejemplo: Este patrón puede verse en las torres de control de los aeropuertos. Los pilotos de los aviones que se encuentran por despegar o aterrizar se comunican con la torre en lugar de hacerlo explícitamente entre ellos. La torre de control regula quien puede aterrizar y despegar, pero no se encarga de controlar todo el vuelo.

18. Memento: preserva una “fotografía instantánea” del estado de un objeto con el fin de permitirle volver a su estado original, sin revelar su contenido al mundo exterior.

    Ejemplo: una funcionalidad muy importante de los editores de texto es “Deshacer” o “Undo”, esta funcionalidad puede implementarse vía Memento. Para realizar esto se debe considerar al contenido del documento como estado del editor y ante cada cambio del documento se debe tomar una nueva fotografía del estado del documento antes de la modificación, para poder volver al mismo. Dos factores importantes deben ser tenidos en cuenta: el orden de guardado de los cambios, para poder deshacerlos correctamente y cuando limpiar el registro de estados intermedios ya que esto puede consumir muchos recursos.

19. Observer: brinda un mecanismo que permite a un componente transmitir de forma flexible mensajes a aquellos objetos que hayan expresado interés en él. Estos mensajes se disparan cuando el

objeto ha sido actualizado, y la idea es que quienes hayan expresado interés reaccionen ante este evento.

Ejemplo: este patrón puede verse en las subastas donde cada ofertante (Observer) tiene un indicador con su número, el cual es utilizado para indicar la aceptación de una oferta. El subastador(Subject, objeto observado) comienza la subasta con una oferta inicial, cuando un ofertante toma esa oferta el subastador les retransmite a todos los ofertantes que el precio ha cambiado.

20. State: permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. El objeto parecerá que cambió de clase.

    Ejemplo: este patrón puede observarse en las máquinas expendedoras de golosinas, las cuales pasan por distintos estados: stock disponible, dinero depositado, capacidad para dar vuelto, golosina seleccionada, etc. En cada uno de éstos estados la máquina se comporta distinta. Cuando se deposita dinero y se elije una golosina la expendedora puede entregar un producto y no cambiar su estado, entregar un producto y cambiar su estado (por ejemplo quedarse sin stock, en cuyo caso no entregará mas golosinas, o no entregar golosinas ya sea por falta de stock o cambio).

21. Strategy: define una jerarquía de clases que representan algoritmos, los cuales son intercambiables. Estos algoritmos pueden ser intercambiados por la aplicación en tiempo de ejecución.

    Ejemplo: Se dispone de un programa que encripta y desencripta mensajes de texto usando distintos algoritmos de encriptación. Cada uno de estos algoritmos de encriptación puede modelarse como una clase con servicios de encriptación (un método “encriptáMensaje” que recibe el texto llano y una clave, para devolver un texto cifrado y servicios de desencriptación (un método “desencriptáMensaje” que recibe una clave y un texto cifrado para devolver uno descifrado).

22. Template Method: define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

23. Visitor: representa una operación sobre elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera. Brinda una forma sencilla y mantenible de realizar acciones sobre una familia de clases.

    Ejemplo: un compilador interpreta código fuente y lo representa como un árbol de sintaxis abstracta (Abstract Syntax Tree, AST), el cual cuenta con diversos tipos de nodos (asignaciones, expresiones condicionales, etc.). Sobre este árbol se desean ejecutar algunas operaciones como: revisar que todas las variables fueron declaradas, chequeos de tipos de datos, generación de código, etc. Una forma de realizar estas operaciones es mediante la implementación del patrón Visitor, el cual recorrerá toda la estructura del árbol. Cuando un nodo acepte al Visitor, éste invocará al método de visita definido en el Visitor que toma por parámetro al nodo siendo visitado.

3. Refactorización

Una refactorización es una transformación controlada del código fuente de un sistema que no altera su comportamiento observable, cuyo fin es hacerlo mas comprensible y de más fácil mantenimiento. Es una forma disciplinada de limpiar el código minimizando las probabilidades de introducir defectos.

Este proceso permite tomar diseños defectuosos, con código mal escrito (duplicidad, complejidad innecesaria, por ejemplo) y adaptarlo a uno bueno, más organizado. También muestra que el diseño no se da solo al inicio, sino también a lo largo del ciclo de desarrollo, durante la codificación, de manera tal que el diseño original no decaiga.

La figura 1 muestra como luego de haber modificado la estructura interna de la caja (al inicio, gris, con formas irregulares, y al final blanca, con figuras regulares) ante la misma entrada (un óvalo negro), se obtiene la misma salida (un pentágono gris). Esto ilustra la modificación de la estructura interna sin haber modificado el comportamiento observable.

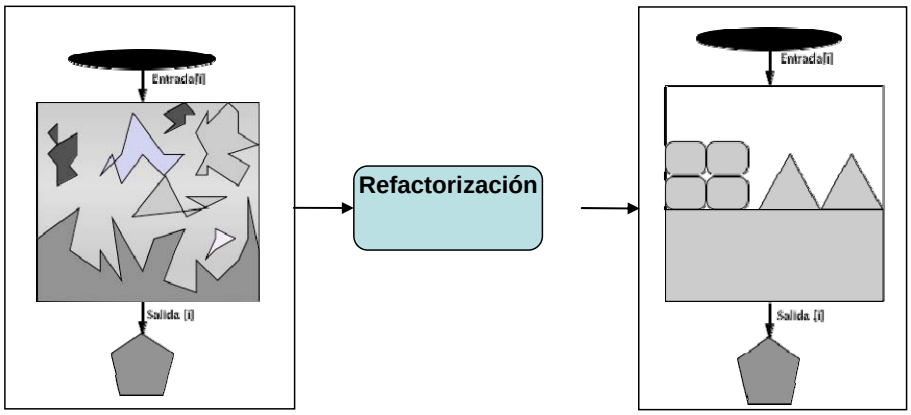

Figura 1. Refactorización: antes y después

3.1. Proceso de refactorizar

Para poder refactorizar de forma satisfactoria es indispensable contar con un buen lote de casos de prueba que validen el correcto funcionamiento del sistema. Estos casos de prueba deben cumplir con los siguientes requisitos:

- Deben ser automáticos de manera tal que se puedan ejecutar todos a la vez con simplemente “hacer clic un botón”;
- Deben ser auto verificables de manera tal de no invertir tiempo en la verificación de los resultados de los mismos. Estos errores deben ser reportados por cada caso de prueba;
- Deben ejecutarse de manera independiente uno del otro, de manera tal que los resultados de uno no afecten los resultados del resto.

Los casos de prueba permiten verificar repetida e incrementalmente si los cambios introducidos han alterado el comportamiento observable del programa.

El primer paso del proceso de refactorización es ejecutar las pruebas antes de haber efectuado cualquier cambio, esto provee información sobre el comportamiento actual del sistema. Éstos resultados deben ser los mismos que se obtengan luego de la refactorización. El segundo paso, consiste en analizar los cambios a realizar, y el tercero es, finalmente, la aplicación del cambio. Como fue mencionado anteriormente se vuelven a ejecutar las pruebas, de manera tal de contrastar los resultados antes y luego de efectuada la

refactorización, los cuales deben ser iguales (Fowler et al, 1999). Al ser iguales los resultados de las pruebas se verifica que no se ha modificado el comportamiento observable del sistema y, por consiguiente, no se han introducido fallas.

Cabe aclarar que una optimización de código no es una refactorización ya que si bien tienen en común la modificación del código fuente sin alterar el comportamiento observable del software, la diferencia radica en la forma de impactar el código fuente: la optimización suele agregarle complejidad.

3.1.1 Importancia de las pruebas en el desarrollo de software

Es importante mencionar la relevancia de las pruebas unitarias para el desarrollo de software ya que realizan otros aportes al margen de "comprobar que el código funciona correctamente", entre los que se encuentran (Massol y Husted, 2003):

- Previenen pruebas de regresión y limitan el debug de código. En primera instancia demuestran que el código fuente funciona correctamente y, también, aportan confianza a la hora de modificar el código ya que se pueden ejecutar nuevamente y verificar si la aplicación del cambio fue correcta. Un buen lote de casos de prueba permite detectar fallas temprano, de manera tal de poder resolverlas con antelación, y reducir el tiempo invertido en la detección de fallas.

- Facilitan la refactorización. Tal como se mencionó con anterioridad, es importante tener casos de prueba que verifiquen el correcto funcionamiento del sistema. El riesgo de introducir una falla luego de una refactorización queda reducido, ya que se pueden detectar a tiempo. Las pruebas unitarias proveen la seguridad suficiente como para facilitar la refactorización.

- Mejoran el diseño de implementación. Es la primera prueba a la que se somete el código fuente, por lo que necesitan que el sistema a probar sea flexible y factible de probar de forma unitaria aislada. Si el código no es fácil de probar de forma unitaria es una señal que indica que debe ser refactorizado.

Si bien los casos de prueba presentan ventajas, también tienen un costo asociado, deben ser mantenidos en línea conforme se modifica el código fuente. Esto significa que si se elimina funcionalidad del sistema deben eliminarse los casos de prueba de dicha funcionalidad, si se agrega funcionalidad se deberán agregar nuevos casos y, si se

modificara funcionalidad existente, las pruebas unitarias deberán ser actualizadas de acuerdo al cambio realizado.

3.2. Aspectos favorables

El proceso de refactorización presenta algunas ventajas, entre las que se encuentran el mantenimiento del diseño del sistema, incremento de facilidad de lectura y comprensión del código fuente, detección temprana de fallos, aumento en la velocidad en la que se programa.

A medida que el software va siendo desarrollado, el equipo de desarrollo va introduciendo cambios, a menudo sin una comprensión profunda del problema a resolver. Esto va ocasionando que el diseño original vaya desvaneciéndose, y sea mas difícil comprender el diseño a partir del código. Éstos cambios van haciendo que de a poco se vaya perdiendo orden en el código y, mientras mas desordenado está, mas difícil es comprenderlo y peor aún mantenerlo. Refactorizar regularmente ayuda a que el código siga representando el diseño.

Una forma importante de mejorar el código es eliminando código duplicado, lo que tiene un impacto directo en las modificaciones futuras del código fuente: mientras mas código hay, mas complicado es modificarlo correctamente ya que hay más código para analizar.

Tanto (Fowler et al, 1999) como (Piattini y García, 2003) coinciden en lo siguiente:

- La refactorización facilita la comprensión del código fuente, principalmente para los desarrolladores que no estuvieron involucrados desde el comienzo del desarrollo. El hecho que el código fuente sea complejo de leer reduce mucho la productividad ya que se necesita demasiado tiempo para analizarlo y comprenderlo. Invirtiendo algo de tiempo en refactorizarlo de manera tal que exprese de forma mas clara cuales son sus funciones, en otras palabras, que sea lo mas auto-documentable posible, facilita su comprensión y mejora la productividad.

- La refactorización permite detectar errores. Cuando el código fuente es más fácil de comprender permite detectar condiciones propensas a fallos, o analizar supuestos desde los que se partió al inicio del desarrollo, que pueden no ser correctos. Mejora la robustez del código escrito.

- La refactorización permite programar mas rápido, lo que eleva la productividad de los desarrolladores. Un punto importante a la hora de desarrollar es qué tan rápido se puede hacer, de hecho un factor clave para permitir el desarrollo rápido es contar con buenos diseños de base.

- La velocidad en la programación se obtiene al reducir los tiempos que lleva la aplicación de cambios: si el código fuente no es fácilmente comprensible, entonces los cambios llevaran mas tiempo. Evita que el diseño comience a perderse. Refactorizar mejora el diseño, la lecto-comprensión del código fuente y reduce la cantidad de posibles fallas, lo que lleva a mejorar la calidad del software entregado, como así también aumenta la velocidad de desarrollo.

3.3. Aspectos desfavorables

Tanto (Fowler et al, 1999) como (Piattini y García, 2003) coinciden en que las áreas conflictivas de la refactorización son las bases de datos, y los cambios de interfaces.

El cambio de base de datos tiene dos problemas: los sistemas están fuertemente acoplados a los esquemas de las bases de datos y el segundo de ellos radica en la migración tanto estructural como de datos. Se deben aplicar los cambios necesarios y luego migrar los datos existentes en la base de datos, lo que es muy costoso.

Uno de los beneficios de la programación orientada a objetos es el hecho de poder cambiar la implementación sin alterar la interface expuesta. El problema del cambio de interface no tiene demasiada complejidad si se tiene acceso al código fuente de todos los clientes de la interface a refactorizar. Sí es problemático cuando ésta se convierte en lo que (Fowler et al, 1999) llama interface publicada (published interface) y no se dispone del código fuente modificable de los clientes de la misma.

3.4. Síntomas que indican la necesidad de refactorizar

Piattini y García (2003) analizan los síntomas que indican la necesidad de refactorizar, a los que Fowler et al (1999) llamaron *bad smells* (malos olores). A continuación se describen brevemente:

1. Duplicated code (código duplicado): Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.

2. Long method (método largo). Legado de la programación estructurada. En la programación orientada a objetos cuando mas corto es un método más fácil de reutilizarlo es.

3. Large class (clase grande). Si una clase intenta resolver muchos problemas, usualmente suele tener varias variables de instancia… lo que suele conducir a código duplicado.

4. Long parameter list (lista de parámetros extensa): en la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino sólo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario. Éste tipo de métodos, los que reciben muchos parámetros, suelen variar con frecuencia, se tornan difíciles de comprender e incrementan el acoplamiento.

5. Divergent change (cambio divergente): una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre si. Este síntoma es el opuesto del siguiente.

6. Shotgun surgery: éste síntoma se presenta cuando luego de un cambio en un determinado lugar, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.

7. Feature envy (envidia de funcionalidad): un método que utiliza mas cantidad de elementos de otra clase que de la propia. Se suele resolver el problema pasando el método a la clase cuyos componentes son más requeridos para usar.

8. Data class (clase de datos): Clases que sólo tienen atributos y métodos de acceso a ellos (“get” y “set”). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.

9. Refused bequest (legado rechazado): Subclases que usan sólo pocas características de sus superclases. Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto suele indicar que como fue pensada la jerarquía de clases no es correcto. La delegación suele ser la solución a éste tipo de inconvenientes.

3.5. Momentos para refactorizar

La refactorización no es una actividad que suele planificarse como parte del proyecto, sino que ocurre bajo demanda, cuando se necesita.

Existe la llamada regla de los tres strikes[†] (Fowler et al, 1999) que sostiene que la tercera vez que se debe realizar un trabajo similar a uno ya efectuado deberá refactorizarse. La primera vez se realiza directamente, la segunda vez se realiza la duplicación y finalmente, a la tercera se refactoriza.

Otros momentos propicios para refactorizar son:

- Al momento de agregar funcionalidad. Es común refactorizar al momento de aplicar un cambio al software ya funcionando, a menudo realizar esto ayuda a comprender mejor el código sobre el que se está trabajando, principalmente si el código no está correctamente estructurado.

- Al momento de resolver una falla. El reporte de una falla del software suele indicar que el código no estaba lo suficientemente claro como para evidenciar la misma.

- Al momento de realizar una revisión de código. Entre los beneficios de las revisiones de código se encuentra la distribución del conocimiento dentro del equipo de desarrollo, para lo cual la claridad en el código es fundamental. Es común que para el autor del código éste sea claro, pero suele ocurrir que para el resto no lo es.

La refactorización ayuda a que las revisiones de código provean más resultados concretos, ya que, no solo se realizan nuevas sugerencias sino que se pueden ir implementando de a poco. Esta idea de revisión de código constante es fuertemente utilizada con la técnica de *pair programming* de *extreme programming*. Esta técnica involucra dos desarrolladores por computadora. De hecho implica una constante revisión de código y refactorizaciones a lo largo del desarrollo.

3.6. Momentos para no refactorizar

Así como existen momentos que son propicios para las refactorizaciones, existen otros que no lo son.

[†] En *baseball* y *softball* un *strike* es un buen tiro del *pitcher*. Luego de tres de ellos el bateador queda fuera. Una analogía a esto podría ser “la tercera es la vencida”.

Cuando se dispone de código que simplemente no funciona, cuando el esfuerzo necesario para hacerlo funcionar es demasiado grande por su estructura y la cantidad aparente de fallas que hacen que sea difícil de estabilizarlo, lo que ocasiona que se deba reescribir el código desde cero. Una solución factible sería refactorizar el software y dividirlo en varios componentes, y luego decidir si vale la pena refactorizar o reconstruir componente por componente.

El otro momento para no refactorizar es cuando se está próximo a una entrega. En este momento, la productividad obtenida por la refactorización misma será apreciable solo después de la fecha de entrega.

4. Antipatrones

Los antipatrones (*antipatterns*) son descripciones de situaciones, o soluciones, recurrentes que producen consecuencias negativas. Un antipatrón puede ser el resultado de una decisión equivocada sobre cómo resolver un determinado problema, o bien, la aplicación correcta de un patrón de diseño en el contexto equivocado.

Según (Brown et al, 1998) “Un antipatrón es una forma literaria que describe una solución recurrente que genera consecuencias negativas”. Analizando parte por parte esta definición se entiende:

- Forma literaria: descripciones de problemas, no de código.
- Recurrente: si no es un patrón, entonces no es un antipatrón. Se deben establecer diversas ocurrencias del mismo comportamiento erróneo preferentemente en diversos contextos.
- Consecuencias negativas: el diseño debe producir impacto negativo.

La esencia de un antipatrón son dos soluciones, en lugar de un problema y una solución como los patrones de diseño. Éstas dos soluciones son: una problemática, que genera consecuencias altamente negativas; y otra llamada refactorizada, en la cual el problema es rediseñado y transformado en una situación mas saludable.

Los antipatrones son una iniciativa de investigación del software que se focaliza en soluciones con efectos negativos, contrario a los patrones de diseño. Esta documentación sobre malas prácticas ayuda a los arquitectos y consultores de software a evitar cometer errores

recurrentes. Sin este conocimiento, los antipatrones seguirán apareciendo en los proyectos de software.

4.1. Relación con patrones de diseño y refactorizaciones

Al igual que los patrones de diseño, los antipatrones, proveen un vocabulario común con el fin de mejorar la comunicación en el equipo de desarrollo, de manera tal de poder identificar problemas y discutir soluciones.

Ambos documentan conocimiento con el fin de ser distribuido para su utilización. Mientras que los patrones de diseño documentan soluciones exitosas, los antipatrones documentan soluciones problemáticas: qué salió mal y por qué (Ang et al, 2005).

Brown et al (1998) sostienen que los antipatrones son el lado oscuro de los patrones de diseño, ya que muchas de las veces que se implementan soluciones basadas en patrones no se evalúa que tan aplicables son para el problema que se está intentando resolver. A su vez, muchos desarrolladores conocedores de patrones de diseño tienden a clasificar todo como capaz de ser resuelto con patrones de diseño, previo a haber finalizado el análisis completo del problema.

4.2. Puntos de vista

Según (Brown et al, 1998) existen tres puntos de vista, o niveles, desde los cuales analizar los antipatrones. Ellos son:

- Desarrollo: describe situaciones encontradas por programadores en la resolución de problemas de programación.
- Arquitectura: estos antipatrones se centran en los problemas recurrentes relacionados a la estructura del sistema, sus consecuencias y soluciones.
- Administración: Parte del trabajo de un gerente implica comunicarse con la gente, y resolver problemas. Los antipatrones de administración identifican los principales escenarios donde éstos problemas pueden llegar a ser destructivos para los proyectos de software.

4.3. Componentes

Los antipatrones están formados por dos componentes: causas principales (*root causes*) y fuerzas principales (*primal forces*). El primer componente, las causas principales, son errores recurrentes presentes

en el desarrollo de software cuyo resultado fue el fallo del proyecto, exceso de costos, atrasos de acuerdo a lo planificado u objetivos de negocio no cumplidos. Las causas principales son las siguientes: prisa (haste), indiferencia (apathy), estrechez mental (narrow-mindeness), pereza (sloth), ignorancia (ignorance), orgullo.

Crawford y Kaplan (2003) sostienen que no solo debe analizarse porqué existen los antipatrones, sino que también se debe analizar porqué persisten y se propagan. A menudo las razones son la inexperiencia, el código ilegible, y el desarrollo basado en “copiar y pegar” código (clonación).

Las fuerzas principales son factores presentes en la toma de decisiones. En una solución de diseño, aquellas fuerzas resueltas satisfactoriamente llevan a beneficios y, aquellas que no, llevan a consecuencias negativas. A continuación se listan las fuerzas principales: administración de funcionalidad, administración de desempeño, administración de complejidad, administración de cambio, administración de recursos de IT, administración de transferencia de tecnología

4.4. Un proceso para el uso de antipatrones

Tate (2002) propone un proceso de seis pasos para el estudio de antipatrones (ver Figura 2):

1. Encontrar el problema: una falla en el comportamiento, un error en el diseño, etc.

2. Establecer un patrón de fallas: identificar en qué condiciones se da el problema.

3. Refactorizar el código: el código que produjo el error debe ser refactorizado y, en la medida de lo posible, la nueva solución debería implementarse utilizando patrones de diseño.

4. Publicar la solución: éste conocimiento debe ser diseminado con el fin de que si otros desarrolladores se encuentran con uno similar podrán resolverlo.

5. Identificar debilidades, o posibles problemas del proceso. A menudo las herramientas inducen el mal uso de las mismas, en otras oportunidades son las presiones externas (por ejemplo plazos de entrega) las que lo hacen. Es importante tener presente que el

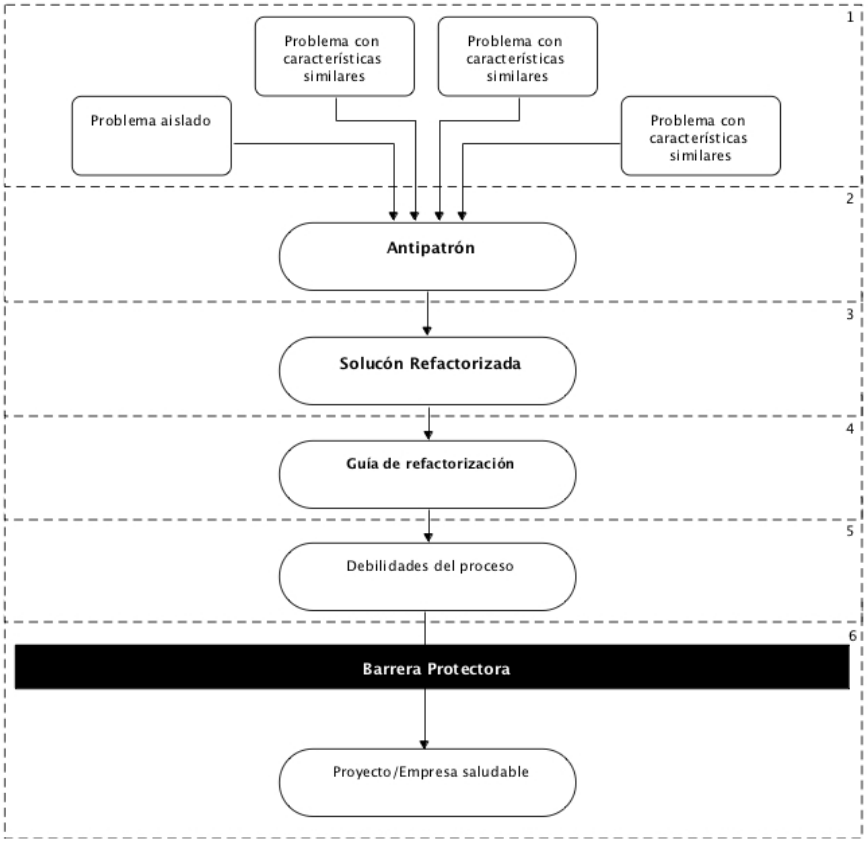

Figura 2: Proceso de aplicación de antipatrones (Tate, 2002)

proceso debe ser realizable por seres humanos imperfectos. En muchos casos, la educación suele ser la solución.

6. Corregir el proceso: Finalmente, una vez identificado y difundido el antipatrón, se construye una barrera entre un problema de iguales características y el proyecto (o empresa), de manera tal que sea posible prevenirlo.

4.5 Catálogo de antipatrones de desarrollo

Los antipatrones de desarrollo de software describen problemas comunes con los que se encuentran los programadores durante el proceso de desarrollo.

Para que un sistema sea fácilmente extensible y mantenible debe estar bien estructurado, lo cual no siempre se logra. A menudo sucede que a medida que se va desarrollando el sistema éste va perdiendo la estructura definida por la arquitectura. Esta estructura va cambiando a medida que los programadores aprenden nuevas restricciones y enfoques que modifican la forma de implementar soluciones. Una forma de volver a estructurar el código fuente es mediante la refactorización.

A continuación se enumeran los antipatrones de desarrollos descriptos por (Brown et al, 1998).

4.5.1. The Blob

The Blob, God Class o Winnebago es una clase, o componente, que conoce o hace demasiado, aparece en aquellos diseños donde una clase monopoliza la mayoría del comportamiento del sistema, mientras que el resto de las clases sólo encapsulan información. Estos diseños son procedurales, incluso cuando son representados con notación del paradigma orientado a objetos y se implementen en un lenguaje orientado a objetos.

El siguiente ejemplo fue tomado de (Brown et al, 1998). Un módulo de GUI que debe ser extendido para interactuar con un módulo de procesamiento gradualmente toma la funcionalidad de los módulos de procesamiento de fondo (en segundo plano, o background). Un ejemplo de esto es una pantalla de PowerBuilder para alta/consulta de clientes. Esta pantalla permite:

- Mostrar información
- Editar información.
- Realizar validaciones simples. Luego el desarrollador agrega funcionalidad que debería agregarse al motor de decisiones: validaciones complejas; algoritmos que utilizan la información validada para evaluar acciones siguientes
- El desarrollador luego recibe los siguientes requerimientos: extender la GUI a tres formularios; soportar scripts (incluyendo el desarrollo de un motor de procesamiento de scripts); agregar nuevos algoritmos al motor de decisiones.

Este ejemplo ilustra cómo un módulo va acaparando nuevas funcionalidades. En lugar de desarrollar varios módulos independientes se actualiza sólo uno. De haber sido diseñado de forma modular hubiese sido más fácil de extender.

4.5.2. Lava Flow

Lava Flow o Dead Code aparece principalmente en aquellos sistemas que comenzaron como investigación o pruebas de concepto y luego llegaron a producción. La principal característica de este antipatrón es la presencia de distintos flujos o corrientes de previos desarrollos que quedaron diseminados, y se hicieron inservibles. Estos desarrollos anteriores (a modo de investigación) a menudo probaban distintos enfoques para resolver distintos problemas, usualmente apresurados para entregar a término para una demostración, omitiendo documentación.

Ejemplo: En un determinado proyecto de desarrollo de decide construir un marco de trabajo (framework) para desarrollar pantallas de la forma mas ágil posible. Este marco de trabajo debe implementarse en Java, y debe permitir desarrollar pantallas de ABM, monitores de consulta, pantallas para previsualizar reportes y a su vez debe permitir también desarrollar otro tipo de pantallas.

A medida que se van realizando las diferentes pruebas (AWT, Swing) el diagrama de clases va creciendo hasta que se obtiene una versión estable del marco de trabajo. Una vez llegado a este punto aquellas ramas del diagrama que no son necesarias no son eliminadas.

La Figura 3 ilustra una jerarquía de clases donde se ven los distintos flujos de desarrollo: una prueba inicial con tecnología AWT, unas primeras pruebas con tecnología SWING, algunas clases sin relación con el resto de la jerarquía, y finalmente la versión estable del marco de trabajo realizado en SWING con el uso de Actions, Listeners y PropertyChangeSupport.

4.5.3. Functional Decomposition

Functional Decomposition o No object oriented AntiPattern. Cuando desarrolladores experimentados en programación estructurada, quienes se hallan cómodos con una rutina principal (main) que llama a diversas subrutinas para realizar el trabajo, diseñan sistemas orientados a objetos suelen traducir sus diseños estructurados a orientados a objetos: traducen cada subrutina como una clase. El resultado de este antipatrón es código estructurado representado en una jerarquía de clases, lo cual suele ser bastante complejo.

El siguiente ejemplo fue tomado de [Brown et al, 1998]. La descomposición funcional se basa en la utilización de funciones para

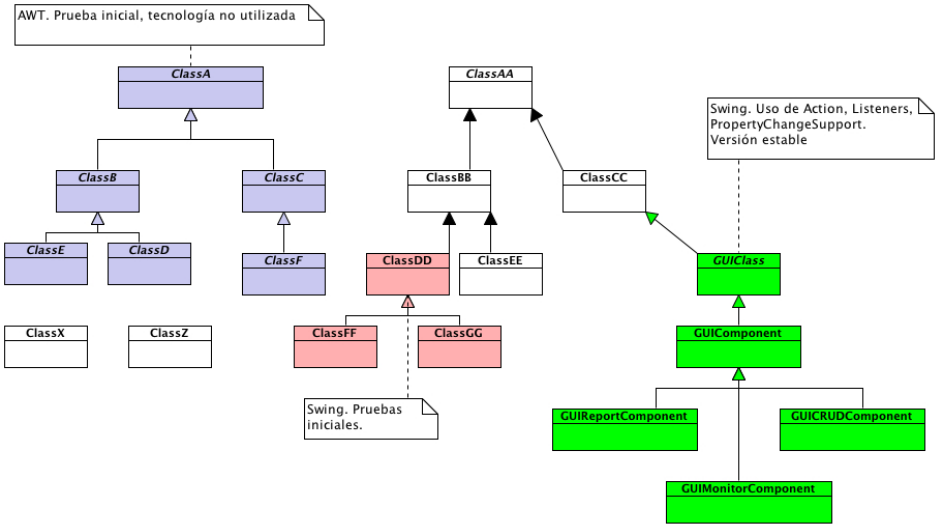

Figura 3. Ejemplo del antipatrón Lava Flow.

manipulación de datos. Las funciones, a menudo, son métodos en un entorno orientado a objetos. Cuando la separación de funcionalidades se realiza mediante distintos paradigmas puede llevar a diversas agrupaciones de funciones y datos. La Figura 4(a) ilustra una versión funcional de un sistema de préstamos a clientes, donde las principales funcionalidades son agregar nuevos clientes; actualizar direcciones de clientes; calcular préstamos a otorgar a clientes; calcular los intereses a cobrar por préstamos; calcular el plan de pagos, modificar el plan de pagos. La Figura 4(b) ilustra la versión orientada a objetos del mismo sistema, donde las funciones fueron mapeadas a métodos.

4.5.4. Poltergeists

Poltergeists, o Gipsy, o Proliferation of Classes, o Big DoIt Controller Class. Las clases poltergeists (fantasmas) se caracterizan por tener pocas responsabilidades dentro del sistema y un ciclo de vida bastante breve, ya que “aparecen” solamente para iniciar algún método en alguna clase, a menudo en un determinado orden. Son de relativa facilidad de encuentro ya que sus nombres suelen llevar el sufijo “controller” o “manager”. Estas clases desordenan el diseño ya que agregan abstracciones innecesarias, son excesivamente complejas, difíciles de mantener y comprender.

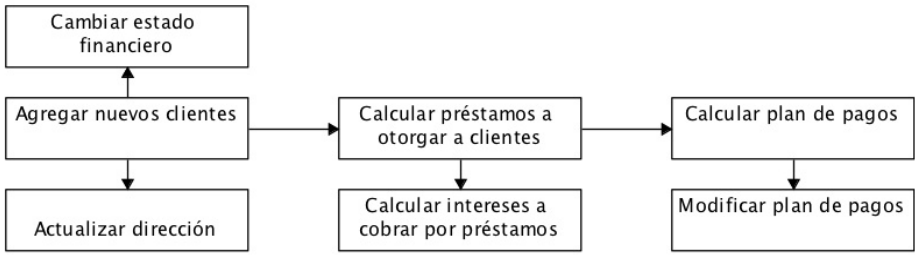

Figura 4(a). Versión funcional del sistema de préstamos

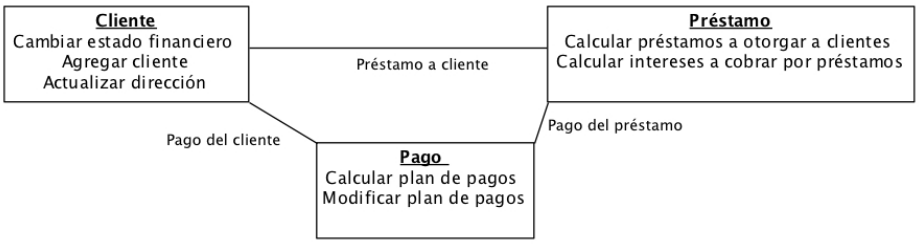

Figura 4(b). Versión orientada a objetos del sistema de préstamos

Ejemplo: La Figura 5(a) ilustra el sistema interno de una máquina de café, en la que se ve que la clase CoffeeMachineController es un Poltergeist porque (i) no tiene estado, (ii) posee relaciones redundantes con el resto de las clases del modelo, y (iii) posee ciclo de vida reducido. La Figura 5(b) ilustra la solución refactorizada, donde se ha eliminado la clase CoffeeMachineController, se han trasladado las responsabilidades de ella a CoffeeMachine y se ha actualizado la jerarquía de clases.

4.5.5. Golden Hammer

Golden Hammer, u Old Yeller, o Head in the Sand. Un martillo de oro (Golden Hammer) es cualquier herramienta, tecnología o paradigma que, según sus partidarios, es capaz de resolver diversos tipos de problemas, incluyo aquellos para los cuales no fue concebido. Este antipatrón tal vez sea uno de los mas comunes de la industria y radica en la creencia de que una tecnología o herramienta realizará mejoras significativas sobre la productividad, reducirá la cantidad de errores y

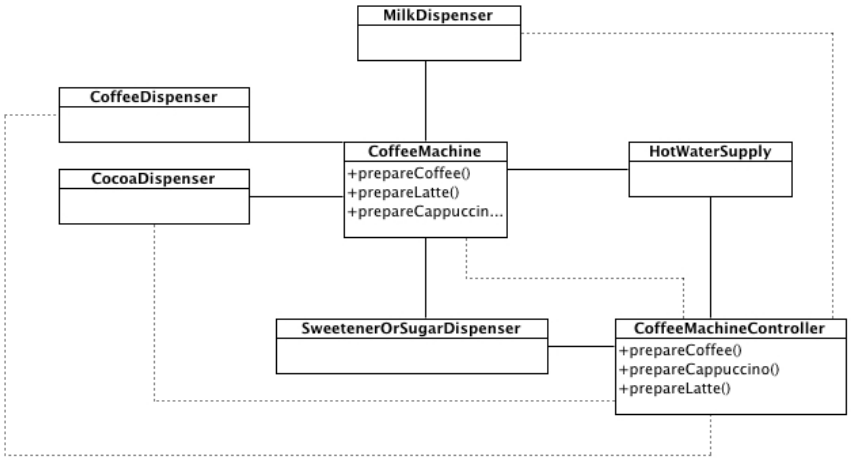

Figura 5(a) Ejemplo Poltergeist

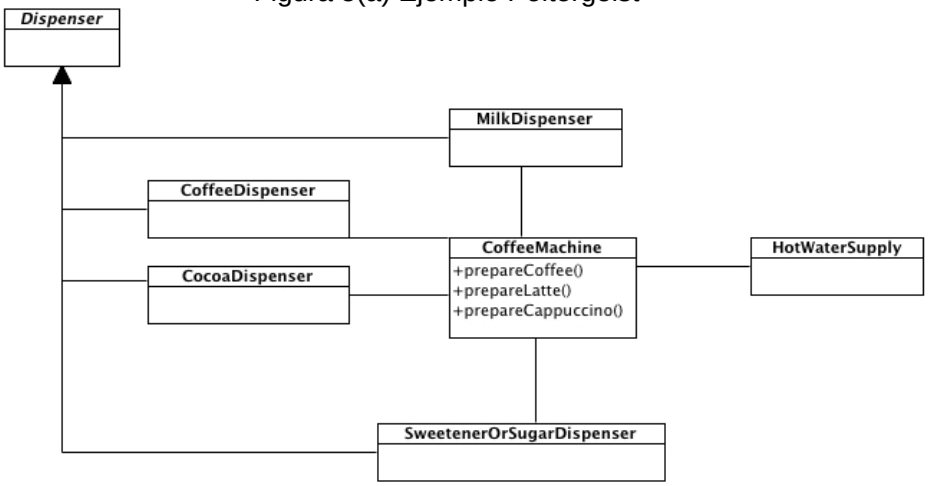

Figura 5(b) Ejemplo Poltergeist refactorizado

disminuirá la cantidad de líneas de código, sin desventajas significativas.

Este antipatrón se da cuando un equipo consigue un alto nivel de competencia en una determinada solución/tecnología (Golden Hammer), lo que hace que cada nuevo desafío que se tenga que afrontar sea mejor resuelto con la herramienta ya conocida. A menudo, el Golden Hammer es un despropósito al problema, pero no se invierte esfuerzo alguno para explorar soluciones alternativas.

Ejemplo: En aplicaciones basadas en procesamiento de mensajes de texto la complejidad es uno de los peores enemigos, para estas aplicaciones XML es el mas reciente Golden Hammer.

Cuando se desarrollan aplicaciones para procesamiento de mensajes de texto el tipo de mensaje tiene un impacto significativo en desempeño y complejidad de la misma. El lenguaje XML posee características que lo hacen interesante: un buen meta-modelo, estructura bien definida y flexibilidad. El problema radica en que el costo y complejidad del mismo puede ser prohibitivo para problemas simples. Es por esto que su uso debería limitarse a circunstancias tales como:

- El bajo acoplamiento es mucho más importante que el rendimiento/desempeño.
- La aplicación hace un fuerte uso de meta-data
- Productores y consumidores de mensajes son aplicaciones distintas, posiblemente con distintos planes de desarrollo y mantenimiento.

4.5.6. Spaghetti Code

Este es el más clásico y famoso antipatrón, el cual pasó de los lenguajes no orientados a objetos a los orientados a objetos. Este antipatrón se manifiesta como un sistema con poca estructura donde los cambios y futuras extensiones se tornan difíciles por haber perdido claridad en el código, incluso para el autor del mismo.

Cuando el sistema está desarrollado en un lenguaje orientado a objetos, este suele incluir pocos objetos con métodos cuyas implementaciones suelen ser extensas que terminan invocando a un solo flujo.

Ejemplo: La emisión de formularios (reportes o informes) agrupan y totalizan información de acuerdo a ciertos criterios dados, por ejemplo un informe de ventas sumará las ventas de los vendedores y las agrupará por mes y vendedor. Un ejemplo de este antipatrón sería la generación del reporte en un solo método donde se realiza todo el proceso:

- Conexión a la base de datos
- Obtención de los datos
- Agrupar y totalizar datos (agrupar por mes y vendedor, sumar ventas)
- Generación de salida resultante

A un archivo

A pantalla
- Cierre de conexión de base de datos.

4.5.7. Copy-And-Paste Programming

Copy-And-Paste Programming, Clipboard coding, Software clonning, o Software propagation. Este antipatrón se basa en la idea de que es más fácil modificar código preexistente que programar desde el comienzo. Se caracteriza por la presencia de fragmentos similares de código diseminados por todo el sistema, que suelen ser modificaciones de otros ya existentes realizadas por desarrolladores poco experimentados. Estos desarrolladores aprenden mediante la modificación de ejemplos producidos por desarrolladores mas experimentados. Mas aún, es mas fácil extender este código ya que el desarrollador tiene control absoluto sobre él, y puede modificarlo para cumplimentar modificaciones a corto plazo de manera tal de satisfacer nuevos requerimientos.

La duplicación de código puede tener efectos positivos a corto plazo, como por ejemplo el incremento en la cantidad de líneas de código, que puede ser utilizado como indicador de desempeño de los desarrolladores.

Ejemplo: Un elemento común a muchos sistemas es el uso de bases de datos para el almacenamiento y obtención de información. Idealmente se debería realizar un componente que abstraiga al programador de cuestiones propias de cada proveedor de base de datos, obtención y cierre de la conexión de base de datos, y que se ocupe solo del problema a resolver una vez obtenida la conexión. Esto es de fácil resolución mediante la implementación del patrón de diseño Template Method, siendo los pasos del algoritmo los siguientes:

1. Obtención de la conexión de base de datos
2. Implementar proceso de negocio que haga uso de la conexión, por ejemplo obtener un listado de ventas. Esta porción del algoritmo debería implementarse como un método abstracto.
3. Cierre de la conexión y manejo de posibles errores.

En un determinado proyecto de desarrollo de software no se tienen en cuenta estos recaudos y el acceso a la base de datos es manejado libremente por los desarrolladores. Los desarrolladores menos experimentados comienzan a copiar y pegar código realizado por los desarrolladores mas experimentados. Durante las pruebas del

sistema se ve que éste comienza a quedarse sin conexiones a la base de datos. El problema: las rutinas de acceso a la base de datos no cerraban las conexiones una vez realizadas las tareas necesarias.

Al haber copiado y pegado esa porción de código por todo el sistema se deberá corregir cada una de las ocurrencias de la falla.

5. Conclusiones

La idea central del uso de Patrones de Diseño y Antipatrones de Desarrollo básicamente consiste en la utilización de técnicas de éxito comprobado y prevención de errores recurrentes, respectivamente.

Existe una ventaja que comparten las tres técnicas mencionadas en el presente artículo, y esta es una mejora cultural, la cual incluye un avance en la comunicación y la documentación. Esto puede sostenerse según el siguiente análisis:

- Los Patrones nominan soluciones exitosas, los Antipatrones situaciones no exitosas y las Refactorizaciones síntomas (“bad smells”). Esta definición de términos comunes permite expresar mas ideas con menos palabras. A modo de ejemplo, cuando se indica que para resolver un problema dado debe aplicarse un determinado patrón, enseguida se recuerdan las clases participantes, colaboraciones típicas y consecuencias.

  Otro factor importante sobre la definición de un vocabulario común está relacionado con la madurez de la disciplina. Otras disciplinas con mayor madurez, por ejemplo Derecho, Química y Física, entre otras, presentan un vocabulario que les es propio.

- Un aspecto a resaltar sobre la documentación es la actitud de colaboración al respecto. A medida que nuevos Patrones y Antipatrones son descubiertos, éstos son publicados y con el tiempo están disponibles a la comunidad, por ejemplo puede encontrarse información sobre Patrones en “Hillside.net – Home Of Pattern Library ”, y sobre Antipatrones en “Source Making ”. Esta documentación contiene información recopilada sobre experiencias exitosas y fallidas, la cual puede ser utilizada con el propósito de mejorar la calidad del software producido.

Esta ventaja de comunicación y documentación representa, en función de lo expuesto en el artículo y lo enunciado precedentemente, el principal beneficio del uso de Patrones y Antipatrones.

Por otra parte estas mejoras en la comunicación tienen un impacto directo en la productividad de los equipos de desarrollo. Éstos reducirán el tiempo de discusiones en las reuniones de diseño gracias al uso de un vocabulario compartido.

En cuanto a la calidad del software producido puede sostenerse que la mantenibilidad (capacidad del producto software para ser modificado (Piatini et al, 2007)) y portabilidad (capacidad del producto software para ser transferido de un entorno a otro (Piatini et al, 2007)) son impactadas por el uso de éstas técnicas:

- Mantenibilidad: esta puede ser mejorada mediante Refactorizaciones ya que su aplicación impactaría en la capacidad del código fuente para ser analizado. La capacidad para ser cambiado se vería impactada por la utilización de Patrones de Diseño ya que éstos agregarían flexibilidad donde sea necesaria.

- Por su parte los Antipatrones permitirían evitar o salir de situaciones infructuosas, tales como la aplicación incorrecta de un patrón de diseño, o la obtención de un diseño menos apropiado por refactorizar de forma incorrecta.

- Portabilidad. Al igual que en la Mantenibilidad, el uso de Patrones de Diseño y Refactorizaciones pueden mejorarla. Aquí el aspecto a mejorar es la adaptabilidad.

Patrones de diseño, Refactorizaciones y Antipatrones representan una mejora en el modo de aplicar el conocimiento de diseño. Es por ello que su aplicación en los procesos de Ingeniería de Software indica un paso en su perfeccionamiento y evolución. Aunque, en algunas oportunidades, la utilización errónea, o el abuso, de Patrones de Diseño puede producir efectos nocivos e ir en detrimento de la calidad y mantenibilidad del software producido. A la hora de aplicar Patrones de Diseño se debe analizar si realmente vale la pena agregar cierta complejidad al diseño en pos de otro tanto de flexibilidad, por lo que se debe ser un consumidor crítico de ellos.

Ante las nuevas metodologías ágiles de desarrollo, en las cuales la tendencia es hacia la simplicidad del diseño, el uso de Patrones de Diseño está casi rechazado en pos de un poderoso proceso de

Refactorización. Aquí el foco está en la semántica, éstas metodologías buscan mantener el código lo más semántico posible, cosa que no necesariamente es factible mediante el uso de Patrones de Diseño donde podrían aparecer clases y colaboraciones que no necesariamente tienen su representación en el negocio.

Respecto de la aplicación de Antripatrones para mejorar procesos en las organizaciones hay que tener especial cuidado en la forma de aplicarlos. Estos no deben ser aplicados en forma insidiosa, buscando responsables a los cuales reprender por su existencia o perduración, sino que deben ser utilizados con el fin de resolver problemas que surgen en torno al desarrollo de software. Debe recordarse que los Antipatrones identifican claramente las malas prácticas para ayudar a evitarlas o resolverlas.

La presencia y perduración de Antipatrones significa una cosa: se están haciendo cosas de forma incorrecta, y cuando esto ocurre es porque no se está tomando la debida responsabilidad en el trabajo. Una forma de evitar la continuidad de los Antipatrones es mediante la motivación hacia los equipos de desarrollo para que reconozcan la existencia de los problemas, para lo cual deben: identificar el problema, determinar qué se está haciendo para resolverlo y qué están haciendo ellos para resolverlo.

Es importante tener presente que, si bien estas tecnologías tienen poca madurez están evolucionando rápidamente y que en un futuro no muy lejano serán un factor casi obligatorio para garantizar la calidad del software. En este futuro, la reinvención de soluciones cederá lugar en favor del crecimiento fundamentado en bases ya afianzadas en Ingeniería de Software

Bibliografía

Ang, J., Cherbakov, L. y Mamdouh, I. *SOA Antipatterns*. www.ibm.com/developerworks/webservices/library/ws-antipatterns, 2005. Página vigente al 2009-06-07.

Brown, W. J., Malveau, R. C. y Thomas J. M. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Estados Unidos, 1998.

Crawford, W. C. R. y Kaplan, J. *JJ2EE Design Patterns*. O'Reilly, Estados Unidos, 2003.

Fowler, M., Beck, K., Brant, J., Opdyke, W. y Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, Estados Unidos, 1999.

Fowler, M. Patterns. *IEEE Software*. Marzo-Abril 2003, 2-3. Disponible en martinfowler.com/ieeeSoftware/patterns.pdf

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. *Patrones de Diseño*. Pearson Educación, Méjico, 1995.

Massol, V. y Husted, T. *JUnit in Action*. Manning Publications, Estados Unidos, 2005.

Piattini Velthuis, M. G. y García Rubio, F. O. *Calidad en el desarrollo y mantenimiento del software*. Alfa Omega Grupo, Méjico, 2003.

Piattini Velthuis, M. G., García Rubio, F. O. y Caballero, I. *Calidad de sistemas informáticos*. Alfa Omega Grupo Editor, Méjico, 2007.

Tate, B. A. *Bitter Java*. Manning Publications, Estados Unidos, 2002.

Vlissides, J. Patterns: The Top Ten Misconceptions. *Object Magazine,* Marzo 1997. Disponible en www.research.ibm.com/designpatterns/pubs/top10misc.html. Página vigente al 2009-06-07.