Universidad Autónoma de Nuevo León
Facultad de Ciencias Físico Matemáticos

Licenciado en Seguridad en Tecnologías de Información

Diseño Orientado a Objetos

Anti-patrones de Diseño

Tarea

Lic. Miguel Ángel Salazar Santillán

Grupo: 007 Matrícula: 1732645

Diana Elizabeth Díaz Rodríguez

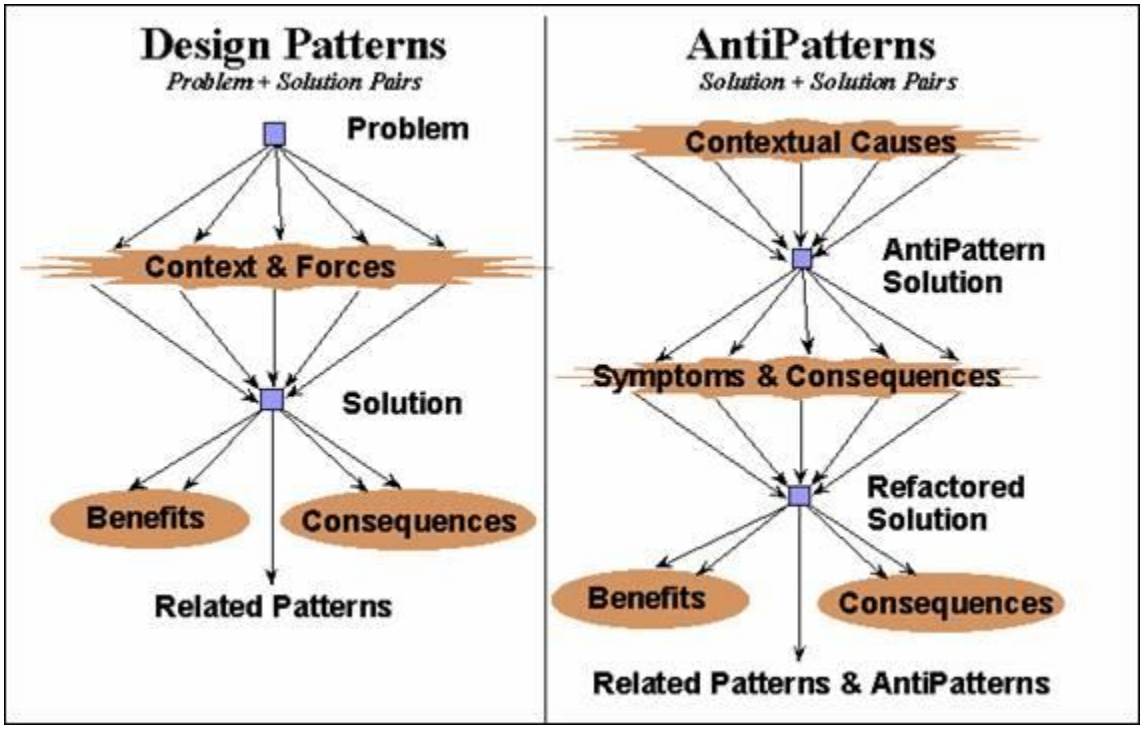
28/ 04/ 2017

Antipattern

Los antipatrones son soluciones negativas que presentan más problemas que los que solucionan. Son una extensión natural a los patrones de diseño. Comprender los antipatrones provee el conocimiento para intentar evitarlos o recuperarse de ellos. El estudio de los antipatrones permite conocer los errores más comunes relacionados con la industria del software.

Los antipatrones se documentan con cierto cinismo, lo cual los hace bastante graciosos y fáciles de recordar. Los nombres siempre aluden al problema que tratan con humor. Se documentan mediante una plantilla (como los patrones de diseño) que incluye secciones para documentar la solución orígen (que es la causa del problema), el contexto, las fuerzas en conflicto y las soluciones correctas propuestas. Un buen antipatrón explica por qué la solución original parece atractiva, por qué se vuelve negativa y cómo recuperarse de los problemas que ésta genera.

Según James Coplien, un antipatrón es...
"...algo que se ve como una buena idea, pero que falla malamente cuando se la implementa."



¿Por qué estudiar antipatrones?

Un antipatrón (antipattern, en inglés) es una forma literaria que describe una solución común a un problema que genera decididamente consecuencias negativas. Según el libro AntiPatterns: Refactoring Software , Architectures and Projects in Crisis, los antipatrones...:

-Son un método eficiente para vincular una situación general a una clase de solución específica.

-Proveen experiencia del mundo real para reconocer problemas recurrentes en la industria del software, ofreciendo también una solución para sus implicaciones más comunes.

-Establecen un vocabulario común para identificar problemas y discutir soluciones.

-Soportan la resolución holística de conflictos, utilizando recursos organizacionales a diferentes niveles.

Categorías de antipatrones

En el libro de antipatrones, éstos se dividen en 3 grandes categorías a las cuales se denominan "puntos de vista":

- *Desarrollo de Software:* Se centran en problemas asociados al desarrollo de software a nivel de aplicación.
- *Arquitectura de Software:* Se centran en la estructura de las aplicaciones y componentes a nivel de sistema y empresa.
- *Gestión de Proyectos de Software:* En la ingeniería del software, más de la mitad del trabajo consiste en comunicación entre personas y resolver problemas relacionados con éstas. Los antipatrones de gestión de proyectos de software identifican algunos de los escenarios clave donde estos temas son destructivos para el proceso de software.

Los anti-patrones, también llamados trampas, son ejemplos bien documentados de malas soluciones para problemas. Se estudian a fin de poderlos evitar en el futuro, y en su caso, para que su presencia pueda ser reconocida fácilmente al investigar sistemas disfuncionales durante una auditoria.

El término anti-patrón se origina como una contra parte al término patrón, acuñado en arquitectura de software, para definir las buenas prácticas de programación, diseño o gestión de sistemas. De tal manera podríamos hablar de que un sistema "bien hecho" está lleno de patrones, y debería carecer de anti-patrones; al menos ese sería un sistema ideal.

Si aplicamos una analogía social, podríamos decir que si a tú, amigo lector, no caes en el anti-patrón (nombrémosle estereotipos) de criminal, terrorista, pervertido, drogadicto, brujo, dictador, y similares, entonces podríamos afirmar que no eres una mala persona. Lo cual no implica, que seas buena; para serlo deberías, no sólo no encajar en los anti-patrones mencionados, sino además, cumplir con uno o más patrones (llamémosle cualidades): honesto, trabajador, buen hijo y/o buen padre, etcétera.

Los anti-patrones en arquitectura de software, son similares a sus análogos sociales, soluciones negativas, acciones que presentan mayores problemas que soluciones. Sin embargo, representan un camino fácil y rápido. Pero continuando con la analogía, podríamos pensar que si necesitas dinero, tienes dos opciones: el patrón (buen comportamiento) trabajar arduamente o el anti-patrón (rápido y con consecuencias a largo plazo) robar un banco.

El mismo concepto se aplica fácilmente en ingeniería, y en otras actividades donde esté presente el esfuerzo humano. Así, anti-patrones como reinventar la rueda, la cosa, el programa Dios, casarse con el diablo, la tubería de la estufa, corn cob (juntos pero no revueltos), etcétera. Dejan la enseñanza de cuál es la mejor forma de no hacer sistemas de cómputo.

En la elaboración de un sistema, intervienen al menos, diversos actores: arquitectos de software, administradores de proyecto y desarrolladores. Para cada uno de ellos, existen anti-patrones que describen comportamientos y soluciones incorrectas Los anti-patrones (una vez conocidos) constituyen para cada uno de los actores involucrados, descripciones de problemas recurrentes en la construcción de software, les proporcionan un vocabulario común para identificar problemas y discutir posibles soluciones y les sugieren pasos para la re-ingeniería, y re-organización estructural de un sistema.

Anti-patrones de Codificación

Revisemos algunas técnicas para codificación incorrecta de software.

***1. Lava Flow*.** Algo así como "programar al estilo volcán". Es construir grandes cantidades de código de manera desordenada, con poca documentación y poca claridad de su función en el sistema. Conforme el sistema avanza en su desarrollo, y crece, se dice que estos flujos de lava se solidifican, es decir, se vuelve mucho más complicado corregir los problemas que originan, y el desorden va creciendo geométricamente. Esto se hace patente cuando: 1. Se declaran variables no justificadas. 2. Se construyen clases o bloques de código muy grandes y complejas sin documentar, o que no se relacionan claramente con la arquitectura. 3. Usando un inconsistente y difuso estilo de evolución de una

arquitectura. 4. Cuando en el sistema existen muchas áreas con código por terminar o reemplazar. 5. Y claro, cuando dejamos código sin uso abandonado; interfaces o componentes obsoletos en el cuerpo del sistema. Los resultados son predecibles: conforme los flujos se endurecen y solidifican (se escribe código y pasa el tiempo), rápidamente se vuelve imposible documentar el código o entender su arquitectura lo suficientemente bien como para hacer mejoras.

2. The God.- Un programa omnipresente y desconocido. Aquel sistema donde una sola clase ó modulo (la función main o equivalente) hace todo. Así que el programa es un solitario y único archivo de muchísimas líneas. En consecuencia, tenemos un código desorganizado y fuertemente interdependendiente.

3. Golden Hammer.- También conocida como la técnica de la barita mágica. Es un vicio relacionado con aferrarse a un paradigma, para solucionar todos los problemas que se nos presenten al desarrollar sistemas, como por ejemplo, siempre querer usar el mismo lenguaje de programación para todos los desarrollos, sea o no conveniente. Es el caso de enamorarnos de .Net, de Java, de PHP. Es importante comprender que cada uno tiene capacidades y limitaciones en aplicaciones particulares. En consecuencia se trata de, uno, el uso obsesivo de una herramienta, y dos, una terquedad de los desarrolladores para usar un paradigma de solución en todos los programas. Lo cual conlleva ocasionalmente a consumir mucho más esfuerzo para resolver un problema.

4. Spaghetti Code.- Se dice de una pieza de código fuente no documentado, donde cualquier pequeño movimiento convulsiona la estructura completa del sistema. En expresión coloquial: codificar con las... los "pies". A diferencia del estilo volcán, donde la crítica es a la forma en que el sistema crece (se anexan módulos), aquí la crítica es a la forma en que se escribe cada una de las líneas, desde la indentación hasta el lenguaje o lenguajes utilizados y su interacción. Ya en el contexto spaghetti, si mezclamos más de un lenguaje de programación en el mismo archivo, el spaghetti es más sabroso. La receta clásica con lenguajes scripts del tipo PHP con HTML y sazonado con JavaScript, ¡es delicioso! (entiéndase un enorme problema). El origen del spaghetti es regularmente un programa creado para hacer una pequeña demostración, que termina, en un dos por tres, trabajando como producto final. Donde está el problema, bien podemos citar lo siguiente como ejemplos: 1. 50% del tiempo de mantenimiento se invierte en entender al sistema original. 2. El spaghetti es causa directa del síndrome del programador desesperado: ¡mejor reescribimos todo el programa! 3. Y obviamente el reuso es imposible. Pero si para colmo, tenemos sólo un chef, o en el contexto un "Programador Solitario". ¿Quién era ese hombre tras el monitor? Que no está disponible para explicarnos su receta. Simplemente se tienen problemas, muchos problemas.

5. Fantasmas.- Demasiadas clases en un programa o tablas en una base de datos. Varias clases o tablas con mínimas responsabilidades. Muchas veces se utiliza para disfrazar la presencia del anti-patrón The God. Se colocan clases inútiles, que disfrazan el hecho que todo el sistema se encuentra construido en uno, o unos cuantos archivos, módulos o clases. Este anti-patrón sugiere un modelo de análisis

y/o diseño inestable: el diseño no coincide con la implementación, y por ello es imposible hacer extensiones al sistema, porque entre tanto "fantasma", encontrar los elementos relevantes es imposible.

Anti-patrones de Arquitectura

Ahora consideremos las malas técnicas para definir componentes y relaciones en el sistema, es decir, su arquitectura. Si los desarrolladores tienen una colección de malas costumbres de codificación (anti-patrones) a su alcance, los arquitectos no nos quedamos atrás. Veamos:

6. Reinventar la rueda.- Se refiere a re implementar componentes que se pueden conseguir prefabricados de antemano, y hacer poco reusó en el código. En breves palabras: querer hacer todo uno mismo. Y hablaríamos de: 1. Poco nivel de reusó en el código, reusó de un proyecto a otro, con lo cual cada proyecto está comenzando desde cero. 2. Constantemente se reescriben fragmentos de código con la misma funcionalidad. 3. Con el consecuente gasto inútil de mano de obra y tiempo en re implementar cosas, que ya estaban hechas. 4. El software se vuelve innecesariamente más denso.

Lo anterior, por lo regular, a causa de poco conocimiento del trabajo ya existente por parte del arquitecto, lo que conlleva a buscar soluciones para problemas ya solucionados.

***7. Casarse con el diablo*.-** Crear una dependencia hacia un fabricante que nos provee de alguna solución (componentes). El problema es inminente: 1. Se depende completamente de lo que el vendedor haga. 2. La calidad de los productos del proveedor nos comprometen. 3. El vendedor nos tiene agarrados. Cito como ejemplo, el caso de una de las universidades más importantes del país, cuyo desarrollo casi completo del sistema de administración escolar, está realizado sobre PL/SQL en Oracle. Ellos necesitan seguir pagando las licencias de Oracle para mantenerse operando. Aún cuando los nuevos desarrollos los estén elaborando ya en otras plataformas Java y PhP, entre otras. Tienen cinco años de desarrollos en PL/SQL, que no los dejan dejar de pagar licencias. No es que Oracle sea malo. Sin embargo, puede ser caro en extremo para una universidad pública.

8. Stovepipe.- Cocinado en "caliente". Es la forma breve de referirse a la creación de islas automatizadas dentro de la misma empresa (cada departamento crea su propio subdepartamento de sistemas). "Islas" independientes, y en conflicto unas con otras. Cada isla desarrolla la parte del sistema que necesita para satisfacer sus requerimientos, sin preocuparse por el resto (no existe un plan o eje guía), el escenario resultante implica una pobre o nula interoperabilidad, obviamente, no existe el reusó y, consecuentemente se incrementan costos. Contrario a lo que se podría suponer, cada vez es más común este tipo de acciones, dada la premura de departamentos específicos dentro de una macro-empresa, por contar con acceso a Tecnologías de Información.

Se habla de un escenario donde prevalece: 1. Una falta de estrategia tecnológica de la empresa. 2. Falta de estándares. 3. Falta de perfil de sistema. 4. Falta de incentivos para la cooperación en el desarrollo de sistemas. 5. Falta de comunicación. 6. Falta de conocimiento sobre los estándares tecnológicos. 7. Falta de interfaces para la integración de sistemas.

Anti-patrones de Administración de Proyecto

Finalmente, los administradores tampoco están exentos de seguir ciertos anti-patrones:

9. The Mythical Month Man.- Mejor conocido como en el entorno como el "súper equipo de programadores". Consiste en la creencia de que asignar más personal a un proyecto, acotará el tiempo de entrega. Regularmente como una forma desesperada de intentar corregir retraso del proyecto. Llega un punto donde entre más personal se asigne, más se retrasa el proyecto.

10. Project Miss-management.- La jefa o el jefe que no saben coordinar. El proyecto se descuida y no se monitorea de manera adecuada, es muy difícil de detectar en etapas iniciales, pero repentinamente emerge de golpe y suele voltear de cabeza la situación del proyecto. Se manifiesta con retrasos en las fechas de entrega y/o áreas incompletas.

11. Corncob.- Los empleados se obstaculizan unos a otros. Personas conflictivas o difíciles de tratar que forman parte del equipo de desarrollo, desvían u obstaculizan el proceso de producción, porque transfieren sus problemas personales o diferencias, con otros miembros del equipo al proyecto. Bajo esta circunstancia, el proyecto no se desarrolla correctamente aunque el personal es bueno.