Antipatrones

**2014**

Alvaro Javier Martinez Cruz

Personal

30/09/2014



**DESARROLLO DE SOFTWARE**

Tabla de contenido

No se encontraron elementos de tabla de contenido.**1**

Escribir el título del capítulo (nivel 2)2

Escribir el título del capítulo (nivel 3)3

**Escribir el título del capítulo (nivel 1)4**

Escribir el título del capítulo (nivel 2)5

Escribir el título del capítulo (nivel 3)6

**The Blob (The God Class)**

**Forma general**

Este antipatrón es encontrado en diseños donde una clase monopoliza el proceso y otras clases encapsulan datos. Se caracteriza por un diagrama de clases compuesto de un **controller** complejo rodeado por simples clases de datos. La clave del problema es que la mayoría de las responsabilidades están ubicadas en una única clase.

En general, el antipatrón The Blob, es un diseño procedural. Incluso puede ser representado usando notaciones de objeto e implementado en lenguajes orientados a objetos. Un diseño procedural separa los proceso de los datos mientras que un diseño orientado a objetos, mezcla los modelos de procesos y datos, junto con divisiones.

The Blob puede ser el resultado de la inapropiada ubicación de requerimientos. Por ejemplo, puede ser un módulo de software al que se le dieron responsabilidades que se superponen con otras partes del sistema para control o gestión del mismo.

La ubicación de responsabilidades no se reparten durante la evolución del sistema, entonces un módulo del sistema se vuelve predominante. A menudo, The Blob es acompañado de código innecesario, haciendo difícil diferenciar entre la funcionalidad útil de la clase Blob y código que ya no es más usado()ver el antipatrón LavaFlow.

Este antipatrón se da en objetos con muchos atributos o muchas operaciones. Esto rompe las ventajas de la programación OO, ya que estas clases tan grandes son muy difíciles de mantener, de reusar y de probar.

**Síntomas y consecuencias**

* Una sola clase con gran cantidad de atributos, operaciones o ambas. Una clase con 60 atributos o más atributos y operaciones usualmente indica la presencia de este antipatrón.
* Una colección dispareja de atributos sin relación y operaciones encapsuladas en una sola clase. Una falta general a la cohesividad de los atributos y operaciones es típico.
* Una sola clase ***Controller*** con simples clases asociadas.
* Una ausencia de diseño orientado a objeto. Un bucle principal del programa dentro de Blob asociado con objetos de datos relativamente pasivos. La única clase ***controller*** a menudo contiene casi la totalidad de la funcionalidad, muy parecido a un programa procedural.
* Un diseño migrado que no ha sido apropiadamente refactorizado a una arquitectura orientada a objetos.
* The Blob compromete las ventajas inherentes de un diseño orientado a objetos. Por ejemplo, limita la habilidad de modificar el sistema sin afectar la funcionalidad de otros objetos encapsulados. Las modificaciones a otros objetos en el sistema también es probable que tengan impacto sobre el software de Blob.
* La clase Blob es típicamente muy complicada de reusar y testear. Puede ser ineficiente o introducir excesiva complejidad reutilizar esta clase para pequeñas partes de su funcionalidad.
* La clase Blob puede ser costosa de cargar en memoria, usando recursos excesivos incluso para operaciones simples.

**Causas típicas**

* *Falta de una arquitectura orientada a objetos.* Los diseñadores pueden no tener un adecuado entendimiento de los principios de orientación a objetos. Alternativamente, el equipo puede carecer de habilidades apropiadas de abstracción.
* *Falta de (cualquier) arquitectura.* La falta de definición de componentes del sistema, sus interacciones y el uso específico de los lenguajes de programación seleccionados. Esto permite que los programas evolucionen de una manera Ad hoc porque los lenguajes de programación son usados para una finalidad que no es la prevista.
* *Falta de cumplimiento de la arquitectura.* A veces, este antipatrón crece accidentalmente, incluso luego de una bien planeada arquitectura. Esto puede ser el resultado de una inadecuada revisión de la arquitectura luego de que el desarrollo toma su lugar. Esto es especialmente frecuente con equipos que son nuevos frente a la orientación a objeto.
* *Intervención demasiado limitada.* En proyectos iterativos, los desarrolladores tienden a agregar pequeñas piezas de funcionalidad a clases existentes que ya están funcionando, en lugar de agregar nuevas clases o revisar la jerarquía de clases para una ubicación más eficaz de las responsabilidades.
* *Desastre especificado.* Algunas veces, The Blob, resulta de la manera en que los requerimientos son especificados. Si el requerimiento dicta una solución procedural, entonces pueden hacerse compromisos arquitectónicos durante el análisis de los requerimientos que luego se hacen difícil cambiar. Definir la arquitectura del sistema como parte del análisis del requerimiento, es usualmente inapropiado y a menudo lleva hacia el antipatrón The Blob, o peor.

**Excepciones conocidas**

El antipatrón The Blob es aceptado cuando se envuelve(Wrapping) un sistema legado. No hay una partición del software requerido, sólo una capa final de código para hacer al sistema legado más accesible.

**Solución refactorizada**

Al igual que la mayoría de los antipatrones, la mayoría involucra una forma de *refactoring*. La clave es mover comportamientos afuera del *Blob.* Puede ser apropiado reubicar comportamientos hacia otras clases que encapsulan datos de forma que estos objetos sean más capaces y *The Bolb* menos complejo.

**Lava Flow ( Dead Code )**

Es comúnmente encontrado en proyectos que iniciaron como una investigación y terminaron siendo el producto final. Se caracteriza porque al que la lava, “fluye” de las versiones anteriores de desarrollo esparcido a lo largo del código, el cual ahora se ha endurecido como el basalto, inamovible, generalmente una masa inútil de código de la que nadie puede recordar mucho.

Este es el resultado de los primeros tiempos de desarrollo cuando, mientras se investigaba, los desarrolladores intentaban muchas formas de cumplir con los requerimientos, típicamente en una carreara para algún tipo de demostración, echando al viento las buenas prácticas y sacrificando la documentación.

El resultado son demasiados fragmentos de código, caprichosas variables de clase y procesos que no estás claramente relacionados al resto del sistema. De hecho, estos flujos son a menudo tan complicados en apariencias y su semejanza al código *Spaghetti* que hasta parecen importantes, pero nadie puede realmente explicar qué hacen o porqué existen.

A veces un viejo desarrollador ermitaño puede recordar ciertos detalles, pero típicamente, todos han decidido “dejar las cosas como estaban” hasta que el código en cuestión “realmente hace daño, actualmente puede ser crítico y no se dispone de tiempo para lidiar con él”.

Aunque puede ser divertido dividir estos flujos y estudiar su “antropología”, usualmente no hay tiempo suficiente en la agenda para semejantes “divagaciones”. En cambio, los desarrolladores por lo general toman una ruta conveniente y trabajan pulcramente alrededor de él.

Este anti patrón es, de cualquier manera, increíblemente común en innovadores departamentos de diseño, en donde las pruebas de código conceptual o prototipo, son rápidamente movidos a producción.

Es un diseño pobre por muchas razones importantes:

* *Lava Flows* son caros de analizar, verificar y probar(test). Todo ese esfuerzo se gasta completamente en vano y es una absoluta pérdida. En la práctica, la verificación y las pruebas son raramente posibles.
* Puede ser costoso al cargar en memoria, perdiendo importantes recursos e impactando en el rendimiento.
* Como con muchos anti patrones, se pierden muchas de las ventajas inherentes de un diseño orientado a objetos. En este caso, se pierde la habilidad de aprovechar la modularización y reutilización sin más proliferación de “burbujas” del *Lava Flow*.

**Síntomas y consecuencias**

* Variables y fragmentos de código en el sistema frecuentemente injustificables
* Segmentos complejos sin documentar, funciones en apariencia importantes o clases que nose relacionan claramente con la arquitectura del sistema
* “Evolución“ muy imprecisa de la arquitectura del sistema
* Bloques enteros de código comentado sin explicación o documentación
* Muchas áreas de código “a ser reemplazadas”
* Código sin usar (muerto), simplemente dejado allí
* Interfaces sin usar, inexplicables u obsoletas en las cabeceras de clases
* Si el código *Lava Flow*  no es removido, puede continuar proliferando como código al ser usado en otras áreas
* Si el proceso que conlleva al *Lava Flow* no es verificado, puede haber un crecimiento exponencial a medida que cada desarrollador, demasiado apurado o intimidado para analizar el flujo original, continúe produciendo nuevos o secundarios flujos a medida que tratan de evitar los originales, empeoran la situación
* A medida que el flujo crece se endurece, se hace rápidamente imposible de documentar el código o entender su arquitectura lo suficiente como para hacer mejoras.

**Causas típicas**

* Código proveniente de la investigación es colocado en producción sin pensar en la gestión de la arquitectura
* Distribución sin control de código sin terminar. La aplicación de muchos enfoques para implementar una funcionalidad
* Un solo desarrollador (lone wolf) escribiendo código
* Falta de gestión de la arquitectura y conformidad con la política de gestión de procesos
* Falta de arquitectura o un desarrollo sin orientación a una arquitectura. Esto es especialmente frecuente en equipos de desarrollo altamente transitorios
* Procesos repetitivos de desarrollo. A menudo, los objetivos del proyecto de software no son claros o cambian repetidamente. Para hacer frente a los cambios el proyecto debe reelaborar, dar marcha atrás y desarrollar prototipos. En respuesta a los plazos de demostración, hay una tendencia a hacer cambios precipitados de código “sobre la marcha” para hacerle frente a los problemas inmediatos. El código nunca es limpiado, dejando la evaluación de la arquitectura y la documentación, aplazadas indefinidamente
* Cicatrices en la arquitectura. A veces, se compromete la arquitectura durante la etapa de análisis de los requerimientos y no se detecta sino hasta después de haber hecho cierta cantidad de desarrollo. Y aunque la arquitectura del sistema puede ser reconfigurada, estos errores en línea son raramente removidos. Puede incluso no ser factible comentar el código innecesario, especialmente en entornos modernos de desarrollo en donde cientos de archivos individuales comprenden la arquitectura del sistema. “¿Quién va a buscar todos esos archivos? Simplemente hay que compilarlos!”

**Excepciones conocidas**

En menor escala, los prototipos del tipo “usar y tirar” en un entorno de investigación son idealmente adecuados para implementar el anti patrón *Lava Flow*. Es esencial para entregar rápidamente y el resultado no es requerido para ser sostenible.

**Solución refactorizada**

Hay sólo una forma segura de prevenir este anti patrón: Asegurarse que las buenas prácticas de arquitectura precedan al desarrollo de código en producción. Esta arquitectura debe ser resguardada por un proceso de gestión que asegure la conformidad arquitectónica y se adapte “la misión”(requisitos cambiantes).

Si la consideración de la arquitectura es omitida, al final, el código que se desarrolla no es parte de la arquitectura a la que apunta y por lo tanto, es redundante o muerto. Con el tiempo, este código muerto se convierte en un problema para el análisis, testing y revisión.

En lo casos en que *LavaFlow* ya existe, la cura puede ser dolorosa. Un principio importante es evitar los cambios de arquitectura durante el desarrollo activo. En particular, esto aplica a las interfaces de software que definen la solución de integración de sistemas. La gestión debe posponer el desarrollo hasta que la arquitectura clara haya sido definida y distribuida a los desarrolladores.

Definir la arquitectura puede necesitar uno o más sistemas de detección de actividades. Un sistema de detección es necesario para localizar los componentes que son realmente usados y necesarios para el sistema. Además, define aquellas líneas de código que pueden ser eliminadas sin problemas. Esta actividad es tediosa; puede requerir las habilidades investigativas de un experimentado detective de software.

Como el código sospechoso de ser *Dead Code* es eliminado, se introducen bugs. Cuando esto sucede, se debe resistir la urgencia de arreglar el síntoma sin comprender la causa del terror. Estudiar las dependencias. Esto ayudará a definir mejor la arquitectura a la que se apunta.

Para evitar el *Lava Flow*, es importante establecer un nivel de interfaces del software que sea estable, bien definido y claramente documentado. La inversión por adelantado en las interfaces de software de calidad puede producir grandes beneficios a largo plazo comparado con el costo de “taladrar” glóbulos endurecidos de lava.

Herramientas tales como las de control de código fuente (SCCS), asisten en la gestión de arquitectura. SCCS está incluido con la mayoría de los entornos Unix y proporcionan una capacidad básica de guardar las historias de actualizaciones a los archivos del proyecto controlado.

**Soluciones relacionadas**

En el mundo competitivo de hoy, a menudo es deseable minimizar el tiempo que se demora entre la investigación y la producción. En muchas industrias, esto es crítico para la supervivencia de una compañía. Donde este es el caso, la inoculación contra *Lava Flow* puede, a veces, ser encontrada en un proceso de gestión de configuración (CM) personalizado que pone ciertos controles de limitación en su lugar en la etapa de creación de prototipos, similar a “hooks” en un verdadero desarrollo de clase de producción sin el impacto de restricción completo en la naturaleza experimental de la investigación.

En donde sea posible, la automatización puede jugar un gran rol, pero la clave yace en la personalización de un cuasi proceso de CM que puede ser fácilmente escalado hacia un sistema de control completo de CM una vez que el producto se mueve hacia el entorno de producción. La cuestión es el equilibrio entre el costo de CM en obstaculizar el proceso creativo y el costo de obtener el control rápidamente del desarrollo una vez que el proceso creativo ha dado a luz algo útil y vendible.

**Aplicabilidad a otros puntos de vista y escalas**

El punto de vista arquitectónico juega un rol clave en la prevención de *Lava Flow* inicialmente. Los administradores pueden también jugar un rol importante al identificar tempranamente *Lava Flows* o las circunstancias que pueden conducir hacia este tipo de anti patrón. Estos administradores deben también tener la autoridad de poner los frenos cuando un *Lava Flow* es identificado por primera vez. Posponiendo además el desarrollo hasta que la arquitectura clara haya sido definida y distribuida.

Como con muchos anti patrones, la prevención es siempre más barata que la corrección, entonces, una inversión inicial en una buena arquitectura y capacitación del equipo pueden por lo general asegurar a un proyecto contra este y muchos otros anti patrones. Si bien este costo inicial no muestra resultados inmediatos, sin dudas es una buena inversión.

**Functional Decomposition**

Este anti patrón, es bueno en un entorno de desarrollo procedural. Incluso resulta útil para comprender la naturaleza modular de una aplicación a gran escala. Desafortunadamente, no se traslada directamente a una jerarquía de clases y aquí es donde comienza el problema.

El anti patrón, es el resultado de experimentados desarrolladores, no orientados a objetos, quienes diseñan e implementan una aplicación en un lenguaje orientado a objetos. Cuando los desarrolladores están cómodos con una rutina “principal” que llama a numerosas subrutinas, puede tener una tendencia de hacer todas subrutinas de una clase, ignorando por completo la jerarquía de clases (y prácticamente ignorando la orientación a objetos). El código resultante se asemeja a un lenguaje estructural como Pascal o FORTRAN en la estructura de una clase. Puede ser increíblemente complejo, como inteligentes desarrolladores procedurales idean formas muy inteligentes de replicar sus métodos probados en el tiempo en una arquitectura orientada a objetos.

**Síntomas y consecuencias**

* Clases con nombres de “funciones” como: Calcular\_Interes o Mostrar\_Tabla pueden indicar la existencia de este anti patrón.
* Una arquitectura increíblemente degenerada que pierde completamente el punto de una arquitectura orientada a objetos.
* Absolutamente ningún aprovechamiento de los principios de la orientación a objetos como la herencia o el polimorfismo. Esto puede ser extremadamente costoso de mantener (nunca se debe subestimar el ingenio de un programador de edad que poco a poco va perdiendo la carrera con la tecnología).
* No hay manera de documentar claramente (o explicar) cómo funciona el sistema. Los modelos de clase no tienen ningún sentido
* No hay esperanzas de obtener jamás reutilización del software
* Frustración y desesperación por parte de los *testers*.

**Causas típicas**

* La falta de entendimiento de la orientación a objetos. Los implementadores no la “comprenden”. Esto es bastante común cuando los desarrolladores cambiar de programar en un lenguaje que no es orientado a objetos a uno que sí lo es. Debido a que hay cambios en la arquitectura, diseño e implementación del paradigma, la orientación a objetos le puede tomar hasta tres años a una empresa para lograrlo plenamente.
* La falta de aplicación de la arquitectura. Cuando los implementadores no tiene idea sobre la orientación a objetos, no importa cuan bien diseñada esté la arquitectura; ellos simplemente no entienden lo que están haciendo. Y sin la supervisión adecuada, por lo general, encontrarán la manera de esquivar algo utilizando las técnicas que ellos conocen.
* Desastre específico. A veces, aquellos que generan las especificaciones y requerimientos, no necesariamente tiene verdadera experiencia con sistemas orientados a objetos. Si el sistema que se especifica asume compromisos arquitectónicos antes del análisis de requisitos, puede, ya menudo lo hace, llevar anti patrones tales como *Functional Decomposition*

**Excepciones conocidas**

Está bien visto cuando una solución que nos sea orientada a objetos es necesaria. Esta excecpción puede ser extendida para tratar con soluciones que son puramente funcionales en naturaleza, pero son envueltas (wrapped) para proporcionar una interfaz orientada a objetos al código de implementación.

**Solución refactorizada**

Si aún es posible determinar cuáles son los requisitos básicos para el software, definir un modelo de análisis para el mismo, para explicar las características críticas desde el punto de vista del usuario. Esto es esencial para descubrir la motivación subyacente para muchos de los fabricantes de software en una base de código particular, la cual ha sido perdida en el tiempo. Para todos los pasos en la solución del anti patrón *Functional Decomposition*, proporciona documentación detallada de los procesos usados como base de los esfuerzos de mantenimientos futuros.

A continuación, formular un diseño que incorpore las piezas esenciales del sistema existente. Sin hacer foco en mejorar el diseño, pero si en establecer una base para explicar del sistema tanto como sea posible.

Idealmente, el diseño justifica, o al menos racionaliza, la mayoría de los módulos de software. Desarrollar un modelo de diseño para código existente es esclarecedor, proporciona una visión de cómo todo el sistema encaja. Es razonable esperar que muchas partes del sistema existen por razones desconocidas y para los cuales no hay especulaciones razonables que puedan intentarse. Examinar el diseño y encontrar subsistemas familiares. Son candidatos a ser reusados. Como parte del mantenimiento del programa, se debe participar de la refactorización del código para reutilizar el código entre subsistemas familiares.

**Soluciones relacionadas**

Si demasiado trabajo ya ha sido invertido en un sistema plagado por *Functional Decomposition*, aún se pueden salvar cosas mediante la adopción de un enfoque similar al enfoque alternativo abordado en el anti patrón *The Blob*.

En lugar de un refactoring de abajo hacia arriba de toda la jerarquía de clases, se podría extender la clase “rutina principal” hacia una clase “Coordinator” que gestione toda o la mayoría de la funcionalidad. Las clases de funciones pueden ser acomodadas dentro de clases cuasi orientadas a objetos, combinándolas y reforzándolas para llevar a cabo su propio proceso en la dirección de la modificada clase Coordinator. Este proceso puede resultar enuna jerarquía de clases que es más “trabajable”.

**Aplicabilidad a otros puntos de vista y escala**

Ambos puntos de vista, el arquitectónico y el administrativo, juegan un rol clave tanto en la prevención inicial como en la vigilancia contínua contra el anti patrón *Funtional Decomposition*. Si una correcta arquitectura orientada a objetos fue planeada inicialmente y el problema ocurre en etapas de desarrollo, entonces el desafío es administrativo para hacer cumplir la arquitectura inicial. Igualmente, si la causa fue una falta general de una incorrecta arquitectura inicial, sigue siendo un reto administrativo el reconocer esto, activar los frenos y obtener ayuda de un arquitecto. Cuanto antes, más barato.

**Poltergeists**

Se tratan de clases que juegan roles y responsabilidades limitadas dentro del sistema; por lo tanto, su ciclo de vida efectivo es bastante breve. Poltergeist desordena el diseño de software, creando abstracciones innecesarias; son excesivamente complejas, difíciles de comprender y difíciles de mantener.

Este anti patrón es típico en casos donde los diseñadores están familiarizados con el proceso de modelado, pero son nuevos en definición de diseños orientados a objetos y definición de arquitecturas. En este anti patrón, es posible identificar una o más apariciones de clases “fantasmales” que aparecen brevemente para iniciar alguna acción en otra clase más permanente. Akroyd llama a estas clases: “Gypsy wagons”. Típicamente, dichas clases fueron creadas como clases *controller* que existen sólo para invocar métodos de otras clases, usualmente, en una secuencia predeterminada. Por lo general son evidentes debido a que sus nombres tienen a menudo el sufijo *\_manager* o *\_controller*.

El anti patrón *Poltergeisters* a menudo intencional por parte de algún arquitecto “nuevo” quien realmente no comprende el concepto de orientación a objetos. Las clases *Poltergeist* constituyen un mal diseño por 3 razones principales:

* Son innecesarias, entonces desperdician recursos cada vez que “aparecen”
* Son ineficientes ya que utilizan varias relaciones redundantes
* Se interponen en el camino de un diseño orientado a objetos apropiado al confundir innecesariamente el modelo de objeto

**Síntomas y consecuencias**

* Relaciones redundantes en la arquitectura
* Asociaciones transitorias
* Clases sin estado
* Clases y objetos temporales, de corta duración
* Clases con una sola operación, que existe únicamente para invocar a otras clases a través de asociaciones temporales
* Clases con nombres de operaciones de tipo “control”, tales como *Iniciar\_ProcesoAlfa*

**Causas típicas**

* *Falta de arquitectura orientada a objetos.* “Los diseñadores desconocen la orientación a objetos”
* *Herramienta incorrecta para el trabajo.* Contrariamente a la opinión popular, un enfoque orientado a objetos no es la solución correcta para todos los trabajos. “No hay forma correcta de hacer las cosas mal”. Es decir, si la orientación a objetos no es la forma correcta de hacer una tarea, no hay forma de poder implementarla correctamente.
* *Desastre específico.* Tal como el anti patrón *The Blob*, la gestión algunas veces compromete a la arquitectura durante el análisis de los requerimientos.

**Excepciones conocidas**

No se conocen excepciones de este anti patrón

**Solución refactorizada**

Los “*cazafantasmas*” resuelven los *Poltergeist* removiendo las clases de la jerarquía por completo. Luego, se debe reemplazar la funcionalidad que “proporcionaba” el *Poltergeist*. Esto es un simple reajuste a la arquitectura para corregirla.

La clave es mover las acciones encapsuladas dentro del *Poltergeist* hacia las clases relacionadas que eran invocadas.

**Soluciones relacionadas**

El “80% de las soluciones” discutidas en el anti patrón *The Bolb* resultan de alguna manera muy similares a *Poltergeist*. La clase *coordinator* presentada aun maneja todo o mucho de la funcionalidad del sistema y típicamente exhibe muchas de las características de un *Poltergeist*.

Aplicabilidad a otros puntos de vista y escalas

Ocurre cuando los desarrolladores están diseñando un sistema como ellos lo implementan aunque ciertamente puede ser el resultado de un diseño inapropiado del sistema. Si este presneta evidencia de que *Poltergeist* es realmente un caso de falla de gestión, queda en manos del lector.

Como con muchos anti patrones de desarrollo, ambos puntos de vista: arquitectónico y administrativo, juegan roles esenciales en la prevención y control contra este. Es a través de un enfoque arquitectónico que un anti patrón es reconocido cuando ni siquiera comienza a emerger. Y a través de una gestión efectiva que es localizado apropiadamente cuando la prevención falla.

Los administradores deberían tener gran cuidado para asegurar que la arquitectura orientada a objetos sea evaluada por calificados arquitectos orientados a objetos tan pronto como sea posible y luego de manera continua prevenir errores de novatos como este anti patrón.

**Golden Hammer**

Este es uno de los anti patrones más comunes de la industria.

“Cuando se posee un martillo de oro, todo parece un clavo”

Muchas veces, un producto de tercero ofrece una solución a un problema. Dado el costo inicial de adoptar esta solución, el proveedor del producto a menudo provee más extensiones de esta solución para otras tecnologías y así reducir costos.

**Generalidades**

Supongamos que un equipo de desarrollo de software ha adquirido un alto nivel en una solución o producto en particular (al que nos referiremos como *Golden Hammer*). Como resultado, todos los nuevos desarrollos son vistos como algo que puede ser resuelto con esta solución. En muchos casos, nuestro *Golden Hammer* no se ajusta a nuestro problema, pero se dedica muy poco tiempo a la investigación de soluciones alternativas.

Este anti patrón conlleva a aplicaciones erróneas de una herramienta, concepto o patrón de diseño. Los desarrolladores y los administradores de proyectos se sienten cómodos con un enfoque existente y reacios a aprender y aplicar uno nuevo que se ajuste mejor.

**Síntomas y consecuencias**

* Herramientas y productos idénticos son usados en una amplia gama de productos diferentes
* Las soluciones tienen baja *performance*, escalabilidad, etc, cuando son comparadas con otras soluciones en la industria
* Los desarrolladores se aíslan de la industria. Demuestran una carencia de conocimiento y experiencia con enfoques alternativos
* Los productos existentes dictan el diseño y la estructura del sistema
* Los nuevos desarrollos dependen en gran medida de un producto o tecnología

**Causas típicas**

* Varios casos de éxito han usado un producto en particular
* Se ha invertido mucho tiempo en conocer o experimentar un producto o tecnología
* El equipo se ha aislado del resto de la industria

**Excepciones conocidas**

Este anti patrón, a veces funciona:

* Si el producto que define las restricciones de la arquitectura es una solución estratégica a largo plazo. Por ejemplo, usar una base de datos Oracle para persistir y procedimientos almacenados para el acceso seguro a los datos
* Si el producto es parte de una suite que proporciona la mayor parte de las necesidades de software

**Solución**

Esta solución involucra tanto un aspecto filosófico como un cambio en el proceso de desarrollo. Filosóficamente, una empresa necesita desarrollar un compromiso con la exploración de nuevas tecnologías. Sin tal compromiso, existe el peligro de depender excesivamente de una tecnología o herramienta especifica. Esta solución tiene dos aspectos: un mayor compromiso por parte de la administración para mejorar profesionalmente a sus desarrolladores conjuntamente con una estrategia de desarrollo que requiere límites específicos del software para permitir migrar de tecnología.

Los sistemas de software necesitan ser diseñados y desarrollados con límites bien definidos que faciliten el reemplazo de componentes individuales de software. Un componente debería aislar el sistema de las características propias de su implementación.

Además, los desarrolladores de software necesitan estar al día sobre las tendencias tecnológicas, tanto en el dominio de la organización como en la industria de software en general. Esto se puede cumplir a través de varias actividades que fomenten el intercambio de conocimientos técnicos. Por ejemplo, los desarrolladores pueden crear grupos para discutir técnicas de desarrollo (patrones de diseño, estándares emergentes, nuevos productos) que puedan impactar en el futuro de la empresa. Finalmente, la administración debe invertir tanto en el desarrollo profesional de los desarrolladores de software como en recompensar a aquellos desarrolladores que tienen la iniciativa de mejorar su propio trabajo.

**Spaghetti Code**

Se trata del anti patrón más famoso. Es un clásico. Ha existido de una u otra forma, desde la invención de los lenguajes de programación. Los lenguajes que no son orientados a objetos, parecen ser más susceptibles a este anti patrón, pero es bastante común entre los desarrolladores que aún tienen que dominar plenamente los conceptos avanzados que subyacen la programación orientada a objetos.

**Generalidades**

El *Spaghetti* *Code* aparece como un programa o sistema que contiene muchas estructuras pequeñas de software. La codificación y extensión progresiva comprometen la estructura del software de tal manera que pierde claridad, incluso para el programador original, si él o ella se encuentran lejos de ese código por un tiempo prolongado.

Si se ha desarrollado utilizando un lenguaje orientado a objetos, el software puede incluir un pequeño número de objetos que contienen métodos con implementaciones realmente grandes. Además, los métodos de los objetos son invocados de una manera muy predecible y hay un grado insignificante interacción entre los objetos del sistema. El sistema es muy difícil de mantener y extender. No hay oportunidad de reutilizar los objetos y módulos en sistemas similares.

**Síntomas y consecuencias**

* Después de extraer código, sólo las partes de los objetos y métodos parecen ser reutilizables. Dividir código *Spaghetti* a menudo puede ser tomada como una mala inversión por parte de la empresa (aunque reporta un beneficio posterior). Por este motivo, debe tenerse muy en cuenta antes de tomar la decisión de la división
* Los métodos que están orientados a procesos; frecuentemente, de hecho, los objetos tienden a tener nombres de procesos
* El flujo de ejecución es determinado por la implementación del objeto y no por los clientes del objeto
* Existe mínima relación entre objetos
* Muchos métodos de objetos no tienen parámetros y utilizan variables de clase o globales para procesar
* El código es difícil de reutilizar, y cuando lo es, a menudo es porque lo han *clonado* (*Copy-Paste*). De cualquier manera, en muchos casos, el código nunca es considerado para reutilizarlo
* Se pierden los beneficios de la orientación a objetos
* El software alcanza rápidamente un punto de rendimiento decreciente; el esfuerzo que involucra el mantenimiento del código existente es mayor que el costo de desarrollar una nueva solución desde cero

**Causas típicas**

* Inexperiencia con las tecnologías de diseño orientado a objetos
* Inexistencia de tutoría
* Revisión de código ineficiente
* No se diseña antes de implementar

**Solución**

El *refactoring* de software o *code cleanup* es una parte esencial de desarrollo. Pero al menos un 70 por ciento, o más, del tiempo debe ser dedicado a la extensión del sistema; por ello es crítico mantener una estructura coherente que soporte dicha extensión.

Cuando la estructura se ve comprometida por tener que soportar necesidades imprevistas, la habilidad del código para soportar extensiones se vuelve limitada y eventualmente, inexistente. El término *code cleanup*, no apunta a los administradores, entonces resulta mejor usar el término *software investment*. Después de todo, el *refactoring* es el mantenimiento de la inversión hecha en el software. Un código bien estructurado tendrá un ciclo de vida mucho más largo y una mejor habilidad para soportar cambios. La mejor manera de resolver este anti patrón es la prevención. Esto es, pensar luego desarrollar un plan de acción para finalmente escribir el código. Ahora, si el código ya está degenerado a un punto tal de estar “inmantenible” y la reingeniería no es una opción, hay que tener en cuenta no profundizar el problema.

* No modifique el código simplemente para agregar una nueva funcionalidad. En lugar de esto, pierda un poco de tiempo realizando un *refactoring* del código exisente haciéndolo un poco más mantenible
* Insistir en un apropiado proceso de análisis orientado a objetos para crear modelos

**Cut and Paste Programming**

La programación mediante cortar y pegar es muy común, pero es una forma degenerada de reutilización de software la cual crea pesadillas a la hora del mantenimiento. Proviene de la idea de que es más fácil modificar código existente que hacerlo desde cero. Esto es generalmente cierto y representa un buen instinto de software. Sin embargo, la técnica puede ser usada de manera excesiva.

**Generalidades**

Este anti parón se identifica por la presencia de una gran cantidad de código desperdigados a lo largo del proyecto. Usualmente, un proyecto posee programadores que están aprendiendo cómo desarrollar software siguiendo ejemplos de desarrolladores más experimentados. Sin embargo, ellos están aprendiendo mediante la modificación de código que se ha demostrado que funciona en situaciones similares y, potencialmente, la personalización permite soportar nuevos tipos de datos o un comportamiento ligeramente modificado. Esto crea código duplicado, el cual puede tener consecuencias positivas a corto plazo, como una aceleración de las métricas de código, las cuales pueden ser usadas en evaluaciones de rendimiento.

**Síntomas y consecuencias**

* Los mismos errores se repiten a lo largo de todo el software a pesar de muchos arreglos locales
* Se incrementan las líneas de código sin añadir verdadera productividad
* Revisiones e inspecciones de código son extendidas innecesariamente
* Se toma difícil localizar y corregir todas las instancias de un error en particular
* Este anti patrón conlleva tener excesivos costos de mantenimiento de software
* Los defectos del software se replican por todo el sistema

**Causas típicas**

* Se necesita un gran esfuerzo para crear código reutilizable y las organizaciones hacen más hincapié en ganancias a corto plazo que en una inversión a largo plazo
* La organización no promueve componentes reusables y la velocidad de desarrollo eclipsa todos los demás factores de evaluación
* La inexistencia de conocimiento de abstracción entre los desarrolladores es a menudo acompañado por un pobre entendimiento de la herencia, composición y otras estrategias de desarrollo
* Los componentes reusables, una vez creados, no son suficientemente documentados o hechos disponibles para los desarrolladores
* El **síndrome** “Aún no está hecho” abunda en el entorno de desarrollo
* La inexistencia de previsión o pro actividad entre los equipos de desarrollo

**Solución**

La clonación ocurre frecuentemente en entornos donde el reuso de tipo *White-box* es la forma predomínate de extensión del sistema. En estos casos, los desarrolladores extienden los sistemas a través de la herencia. Ciertamente, la herencia es una parte esencial del desarrollo orientado a objetos, pero tiene severos inconvenientes en sistemas grandes.

Primero, derivar y extender un objeto requiere cierto conocimiento de cómo el objeto está implementado así como las restricciones y patrones de uso indicados por las clases base heredadas. Muchos lenguajes de programación orientados a objetos imponen algunas restricciones, tipo s de extensiones pueden ser implementados en una clase derivada y llevar un uso inadecuado de derivación. Además, por lo general, el reuso por caja blanca es posible solamente en tiempo de compilación de la aplicación así como todas las subclases deben estar completamente definidas antes de que la aplicación sea generada.

Por otro lado, el reuso por caja negra (*black-box*) tiene un diferente conjunte de ventajas y limitaciones. Frecuentemente es una mejor opción para la extensión de objetos en sistemas de moderado y gran tamaño. Con este reuso un objeto es usado tal cual es, usando su interfaz pública. El cliente no tiene permitido alterar cómo el objeto está implementado. El beneficio clave del reuso de caja negra es que, con el soporte de herramientas, tales como lenguajes de definición de interfaces, la implementación de un objeto puede ser llevada a cabo independientemente de la interfaz del objeto. Esto posibilita a un desarrollador para tomar la ventaja de un enlace tardío mediante la asignación de una interfaz con su implementación en tiempo de ejecución. Los clientes pueden ser escritos en base a una interfaz de objeto estático y aún beneficiarse por el tiempo mediante servicios más avanzados que soportan una idéntica interfaz de objeto.

La diferencia entre la reutilización de caja blanca y caja negra, es la misma que hay entre la POO y la programación orientada a componentes COP, donde la derivación de clases de tipo caja blanca es la firma tradicional de OPP y el enlace en tiempo dinámico de interfaces a una implementación es un elemento básico de COP.

Restructurar el software para reducir o eliminar clones requiere la modificación de código para enfatizar el reuso de caja negra de los segmentos duplicados del código. En el caso en que la programación mediante *Cut and Paste* ha sido extensivamente a lo largo de la vida del proyecto, el método más efectivo de recuperar la inversión es refactorizar el código base en librerías reusables o en componentes que hagan foco en el reuso, mediante caja negra, de la funcionalidad. Si se lleva a cabo como un único proyecto este proceso de refactoring es generalmente difícil, prolongado y costoso. Requiere de un arquitecto para supervisar y ejecutar procesos, y mediar en las discusiones sobre las ventajas y limitaciones de las diferentes maneras de extender los módulos. Un refactoring efectivo para eliminar el código repetido involucra 3 etapas: *code mining, refactoring* y *configuration management*. La primera es la identificación sistemática de las múltiples versiones de los mismos segmentos de código. La segunda involucra el desarrollo de una versión estándar de dichos segmentos y reinsertarlo en el código base. Finalmente, *configuration management* son un conjunto de políticas elaborado para ayudar a prevenir futuras ocurrencias de este anti patrón. En su mayor parte, esto requiere de políticas de supervisión y detección (inspecciones de código, revisión, validación) además de los esfuerzos educativos.

**Mini Anti Patrones**

**Continuous Obsolescence**

**El problema**

La tecnología cambia tan rápidamente que los desarrolladores tiene problemas para mantener el paso con las versiones actuales de software y encontrar combinaciones de productos que puedan trabajar juntos.

Dado que cada línea comercial de producto evoluciona con cada nuevo *reléase*, la situación se vuelve cada vez más difícil para que los desarrolladores le hagan frente. Encontrar *releases* de productos compatibles que interoperen exitosamente, es aún más difícil.

Java es un conocido ejemplo de este fenómeno, con nuevas versiones saliendo cada pocos meses. Por ejemplo, en el momento que un libro de java 1.x va la imprenta, un nuevo *Java DevelopmentKit* obsoleta esa información. Java no es el único, muchas tecnologías participan en *Continuous Obsolescence.*

Los ejemplos más flagrantes son aquellos productos que incluyen el año en su marca, tal como Product98. De esta manera, estos productos ostentan la progresión de su obsolescencia. Otro ejemplo es la progresión de las tecnologías dinámicas de Microsoft:

* DDE
* OLE 1.0
* OLE 2.0
* COM
* ActiveX
* DCOM

Desde la perspectiva del vendedor, hay dos factores clave: *mindchara*  y *marketchare*. La innovación rápida requiere atención especializada de los consumidores para estar al día con las últimas características de productos, anuncios y terminología. Para aquellos que siguen la tecnología, la rápida innovación contribuye a *mindshare*; en otras palabras, siempre hay novedades al respecto de tecnología X. Una vez que un *marketshare* es obtenido, los ingresos principales de los proveedores es a través de la obsolescencia y el reemplazo de *releases* de productos anteriores. Cuanto más rápido la tecnología se vuelve obsoleta (o es percibida como tal), más alto es el ingreso.

**Solución refactorizada**

Un importante factor de estabilización en el mercado de la tecnología se basa en estándares de sistemas abiertos (*open systems standars*). Un estándar de consorcio es el producto del consenso industrial que requiere tiempo e inversión.

Las iniciativas conjuntas de marketing crean conciencia y aceptación por parte de los usuarios de cómo la tecnología se mueve dentro de la corriente principal. Existe una inercia inherente en este proceso que beneficia al consumidor, porque una vez que el vendedor establece un estándar para su producto, es improbable que el fabricante cambie las características que conforman al producto.

Las ventajas de una tecnología que se vuelve obsoleta rápidamente, son transitivas. Arquitectos y desarrolladores deben depender de interfaces que son estables o sobre las que ellos tienen control. Los estándares de sistemas abiertos brindan una medida de estabilidad a un mercado que resulta ser caótico.

**Variaciones**

El antipatrón *Wolf Ticket Mini* describe variso enfoques que los consumidores pueden usar para infuir sobre la dirección de un producto para dirigirlo hacia un producto con mayor calidad.

**Ambiguos Viewpoint**

**Problema**

El análisis y diseño de los modelos orientados a objetos (OOA&D) son a menudo presentados sin poner en claro el punto de vista presentado por el modelo. Por defecto, los modelos OOA&D indican un punto de vista de implementación que es potencialmente, como mínimo, útil. Puntos de vista mezclados no permiten la separación fundamental de las interfaces de los detalles de su implementación, lo cual es uno de los beneficios primarios del paradigma de la orientación a objetos.

**Solución refactorizada**

Existen tres puntos de vista fundamentales para los modelos OOA&D: el punto de vista del negocio, el punto de vista de la especificación y el punto de vista de la implementación. El primero define el modelo del usuario de la información y los procesos. Es un modelo que los expertos de dominio pueden defender y explicar (comunmente llamado modelo de análisis). Los modelos de análisis son algunos de los modelos más estables del sistema de información y valen la pena mantener.

Los modelos pueden ser menos útiles si no se enfocan en la perspectiva necesaria. Una perspectiva aplica filtros a la información. Por ejemplo, definir un modelo de clase para un sistema de intercambio telefónico puede variar significativamente dependiendo del enfoque proporcionado por las siguientes perspectivas:

* Usuario telefónico: se preocupa por la facilidad de hacer llamadas y recibir facturas detalladas.
* Operador telefónico: se preocupa por conectar usuarios con los números solicitados.
* Departamento de cuentas telefónicas: quien se preocupa por las fórmulas para la facturación y los registros de todas las llamadas realizadas por los usuarios.

Algunas de las mismas clases serán identificadas, pero no muchas; donde hay, los métodos no serán los mismos.

El punto de vista de la especificación hace foco en las interfaces del software. Dado que los objetos (como tipos de datos abstractos) tienen la intención de ocultar los detalles de implementación detrás de interfaces, el punto de vista de la especificación define las abstracciones expuestas y los comportamientos en un objeto de sistema. Define el límite entre los objetos del sistema.

El punto de vista de la implementación define los detalles internos de los objetos. Los modelos de implementación son a menudo llamados modelos de diseño en la práctica. Al ser un modelo preciso del software, los modelos de diseño deben ser mantenidos continuamente a medida que el software es desarrollado y modificado. Puesto que un modelo desactualizado es inútil, sólo los modelos seleccionados son oportunos de mantener; en particular, aquellos modelos de diseño que representan aspectos complejos del sistema.

**Walking through a Minefield**

Gran cantidad de bugs son encontrados en los productos liberados; de hecho, los expertos estiman que el código fuente original contiene entre dos y cinco bugs por línea de código. Esto significa que el código requerirá dos o más cambios por línea para quitar todos los defectos. Sin duda, muchos productos son liberados mucho antes de que estén listos completamente. Un reconocido ingeniero de software sostiene que "No hay sistemas nuestros, ni siquiera los nuestros".

La ubicación y las consecuencias del software defectuoso no tiene relación con sus causas aparentes, incluso una cantidad mínima de bugs pueden resultar catastróficos. Por ejemplo, los sistema operativo (UNIX, Windows, etc.) contienen gran cantidad de defectos de seguridad conocidos y desconocidos que los vuelven vulnerables a ataques; además, Internet ha incrementado dramáticamente la probabilidad de ataques al sistema.

Los usuarios finales encuentran bugs frecuentemente. Por ejemplo, aproximadamente 1/7 números de teléfonos discados no son completados por el sistema telefónico. Y hay que notar que la tasa de quejas es baja en comparación con la frecuencia de las fallas del software.

El propósito de probar el software comercial es limitar el riesgo, en particular, para soportar los costos de productos empaquetados, cada que un usuario final contacta a un vendedor para solicitar soporte técnico, gran parte (o todo) el margen de ganancia es gastado en responder a esa llamada.

**Solución**

Se requiere una inversión apropiada en pruebas del software para hacer que el sistema se encuentre relativamente libre de bugs. En algunas compañías "avanzadas", el equipo de testing es mayor que el equipo de desarrollo. El cambio más importante para hacer las pruebas es el control de la configuración de los casos de prueba.

Un sistema típico puede requerir hasta cinco veces más software de prueba que software en producción. A menudo, el software de prueba es más complejo que el software de producción debido a que incluye la administración explícita de la sincronización de la ejecución para detectar muchos bugs.

Cuando una prueba detecta un bug, es más probable que sea el resultado de un bug en la prueba que en el código que está siendo probado. El control de la configuración posibilita la administración de los recursos del test, por ejemplo, soportar tests de regresión.

Otros enfoques efectivos para testing es la automatización de la ejecución de test y su diseño. La ejecución manual de las pruebas es un trabajo intensivo y no hay bases que prueben la efectividad de los test manuales.

En contraste, la ejecución automática de las prueba posibilita correrlos con el ciclo de build. Los tests de regresión pueden ser ejecutados sin intervención manual, asegurando que las modificaciones del software no causan defectos en los comportamientos previamente probados.El diseño automatizado de tests soporta la generación de rigurosas baterías de pruebas y existen docenas de herramientas que lo posibilitan.

**Mushroom Management**

**Boat Anchor**

**Dead End**

**Input Kludge**

**ARQUITECTURA**

**Stovepipe Enterprise**

**Vendor Lock-in**

**Architecture by Implication**

**Design by Committee**

**Reinvent the Wheel**

**MINI ANTIPATTERNS**

**Aurogenerated Stovepipe**

**Jumble**

**Cover your Assets**

**Wolf Ticket**

**Warm Bodies**

**Swiss Army Knife**

**The Grand Old Duke of York**

GESTIÓN

**Analysis Paralysis**

**Death by Planning**

**Corncob**

**Irrational Management**

**Project Missmanagement**

**MINI ANTIPATTERNS**

**Blowhard Jamboree**

**Viewgraph Engineering**

**Fear of Success**

**Intellectual Violence**

**Smoke and Mirrors**

**Throw it over the wall**

**Fire Drill**

**The Feud**

**E-Mail is Dangerous**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |