Universidad Autónoma de N.L

Facultad de Ciencias Físico Matemáticas

Investigación AntiPatterns

Diseño Orientado a Objetos

Nombre: Jesús Eduardo Alvarado Ramos.

Grupo: 007

Matricula: 1726329

Viernes 28 de abril del 2017

El **AntiPatterns** como su diseño explica modela a colegas, define un vocabulario de industria para los procesos comunes defectuosos y puestas en práctica dentro de organizaciones. Un vocabulario de nivel más alto simplifica la comunicación entre médicos de software y permite la descripción concisa de conceptos de nivel más alto.

Un **AntiPatterns** es una forma literaria que describe una solución que ocurre comúnmente con un problema que genera consecuencias decididamente negativas. El **AntiPatterns** puede ser el resultado de un director o el revelador no que conoce cualquier mejor, no teniendo el conocimiento suficiente o experimentar en la solución de un tipo particular de problema, o habiendo aplicado un modelo absolutamente bueno en el contexto incorrecto.

El **AntiPatterns** proporciona la experiencia verdadera mundial en el reconocer problemas que se repiten en la industria de software y proporciona un remedio detallado por los apuros más comunes. El toque de luz de **AntiPatterns** los problemas más comunes que afrontan la industria de software y proporcionan los instrumentos para permitirle reconocer estos problemas y determinar sus causas subyacentes.

Además, el **AntiPatterns** presenta un plan detallado para invertir estas causas subyacentes y realización de soluciones productivas. El **AntiPatterns** con eficacia describe las medidas que pueden ser tomadas en varios niveles para mejorar el desarrollo de usos, el diseño de sistemas de software, y la dirección eficaz de proyectos de software.

Ejemplos:

Bikeshedding

De vez en cuando, nosotros interrumpiríamos esto para hablar de la tipografía y el color de la cubierta. Y después de cada discusión, nos pidieron votar. ¡Pensé sería lo más eficiente votar a favor del mismo color que nosotros habíamos decidido en la reunión antes, pero se resultó que yo era siempre en la minoría! Finalmente escogimos rojo. (Esto salió azul. ¿) Richard Feynman, Qué Se preocupa Usted Lo que Otra Gente Piensa?

What is it?

Tendencia de gastar debate de cantidades de tiempo excesivo y decidiendo de publicaciones triviales y a menudo subjetivas.

Why it's bad

Esto es una basura de tiempo. Poul-Henning Kamp entra en la profundidad en un correo electrónico excelente aquí. excellent email here.

How to avoid it

Anime a miembros de equipo a ser consciente de esta tendencia, y priorizar el alcance de una decisión (el voto, tirar una moneda, etc. si usted tiene a) cuando usted lo nota. Considere A/B pruebas más tarde para visitar de nuevo la decisión, cuando es significativo hacer así (p.ej. decidiendo entre dos diseños de UI diferentes), en vez del remoto debate interno.

Examples and signs

Gastos de horas o días discutiendo sobre lo que el fondo colorea para usar en su app, o si hay que poner un botón a la izquierda o el derecho del UI, o usar etiquetas en vez de espacios para la mella en su base de código.

The tricky part

Bikeshedding es más fácil para notar e impedir en mi opinión que la optimización prematura. Solamente trate de ser consciente de la cantidad de tiempo gastada en la toma de una decisión y el contraste que con como trivial la publicación es, e intervenir si fuera necesario.

tl; dr

Evite gastar demasiado tiempo para decisiones triviales.

God Class

Simple es mejor que complejo. Tim Peters, el Zen de Pitón

What is it?

Las clases que controlan muchas otras clases y tienen muchas dependencias y muchas responsabilidades.

Why it's bad

¿Las clases de Dios tienden a crecer al punto de hacerse pesadillas de mantenimiento? porque ellos violan el principio de responsabilidad sola, ellos son difíciles a la prueba de unidad, el ajuste, y el documento.

How to avoid it

Evite tener clases se convierten en clases de Dios por rompiendo las responsabilidades en más pequeñas clases con una responsabilidad sola claramente definida, probada por unidad, y documentada. También mirar " el Miedo de Adición de Clases " debajo.

Examples and signs

Busque nombres de clase que contienen "al director", "el regulador", "el conductor", "el sistema", "o el motor". Esté sospechoso hacia clases que importan o dependen de muchas otras clases, controlan demasiadas otras clases, o tienen muchos métodos que realizan sin relaciones tareas. Las clases de Dios saben de demasiadas clases y/o controlan demasiado.

The tricky part

Como la edad de proyectos y exigencias y el número de ingenieros crecen, clases pequeñas y bien intencionadas se convierten en clases de Dios despacio. La nueva factorización tales clases puede hacerse una tarea significativa.

tl; dr

Evite clases grandes con demasiadas responsabilidades y dependencias.

Premature Optimization

Nosotros deberíamos olvidar la pequeña eficacia, decir aproximadamente el 97 % del tiempo: la optimización prematura es la raíz de todos los males. Aún nosotros no deberíamos renunciar nuestras oportunidades en esto el 3 % crítico. Donald Knuth, Aunque nunca sea a menudo mejor que *right* ahora. Tim Peters, el Zen de Pitón

What is it?

La optimización antes de que usted tiene bastante información para hacer conclusiones cultas sobre dónde y cómo hacer la optimización.

Why it's bad

Es muy difícil de saber exactamente que será el embotellamiento en la práctica. El intento de optimizar antes del teniendo datos empíricos probablemente termina por aumentar la complejidad de código y el espacio para bichos con mejoras insignificantes.

How to avoid it

Priorize la escritura el código limpio y legible que trabaja primero, usando algoritmos sabidos y probados e instrumentos. Use instrumentos copiadores cuando necesario encontrar embotellamientos y optimizar las prioridades. Confíe en medidas y no conjeturas y especulación.

Examples and signs

Caching antes copiador encontrar los embotellamientos. La utilización "de la heurística" complicada e improbada en vez de un sabido matemáticamente corrige el algoritmo. El escogimiento de un marco nuevo e improbado experimental de web

que teóricamente puede reducir la latencia de petición bajo cargas pesadas mientras usted está en tempranas etapas y sus servidores es ocioso la mayor parte del tiempo.

The tricky part

La parte difícil sabe cuándo la optimización es prematura. Es importante planificar por adelantado para el crecimiento. El escogimiento de diseños y plataformas que tendrán la optimización fácil en cuenta y el crecimiento es clave aquí. Es también posible usar " la optimización prematura " como una excusa para justificar la escritura mal cifra. Ejemplo: escribiendo una O (n2) el algoritmo para solucionar un problema cuando un más simple, matemáticamente corrija, la O (n) el algoritmo existe, simplemente porque el algoritmo más simple es más difícil de entender.

tl; dr

Perfil antes de optimización. Evite negociar la simplicidad para la eficacia hasta que sea necesario, apoyado por pruebas empíricas.

Magic Numbers and Strings

Explícito es mejor que implícito. Tim Peters, el Zen de Pitón

What is it?

Utilización de números sin nombre o literales de cuerda en vez de constants llamado en código.

Why it's bad

El problema principal es que la semántica del número o el literal de cuerda parcialmente o completamente es ocultada sin un nombre descriptivo u otra forma de anotación. Esto hace la comprensión el código más difícil, y si se hace necesario cambiarse el constante, buscar y sustituir u otros instrumentos de nueva factorización pueden introducir bichos sutiles. Considere el pedazo siguiente de código: def create_main_window (): ventana = Ventana (600, 600) * etc....

¿Cuáles están los dos números allí? Asuma el primero es la anchura de ventana y el segundo en la altura de ventana. Si alguna vez se hace necesario cambiar la anchura a 800 en cambio, una búsqueda y sustituir sería peligroso ya que esto cambiaría la altura en este caso también, y quizás otras presencias del número 600 en la base de código. Los literales de cuerda podrían parecer menos propensos a estas publicaciones, pero el teniendo literales de cuerda sin nombre en el código hace la internacionalización más difícil, y puede introducir publicaciones similares para hacer con los casos del mismo literal que tiene la semántica diferente. Por ejemplo, los homónimos en inglés pueden causar una publicación similar con la búsqueda y sustituir; considere dos presencias "de punto", el que en el cual esto se

refiere a un sustantivo (como en " ella tiene un punto ") y el otro como un verbo (como en " para indicar las diferencias ... ").

How to avoid it

El empleo llamó constants, métodos de recuperación de recurso, o anotaciones.

Examples and signs

El ejemplo simple es mostrado susodicho. Este anti modelo particular es muy fácil para descubrir (excepto unos casos difíciles mencionados debajo.)

The tricky part

Hay un área estrecha gris donde puede ser difícil de contar si ciertos números son números mágicos o no. Por ejemplo, el número 0 para lenguas con incluir en un índice a base de cero. Otros ejemplos son el empleo de 100 para calcular porcentajes, 2 para comprobar para la paridad, etc.

tl; dr

Evite tener números inexplicados y sin nombre y ensartar literales en el código.

Fear of Adding Classes

Escaso es mejor que denso. Tim Peters, el Zen de Pitón

What is it?

La creencia que más clases necesariamente hacen diseños más complicados, conduciendo a un miedo de adición de más clases o rotura de clases grandes en varias más pequeñas clases.

Why it's bad

La adición de clases puede ayudar a reducir la complejidad considerablemente. Imagine una pelota grande enredada de hilos. Cuando desenredado, usted tendrá varios hilos separados en cambio. Asimismo, vario simple, " fácil mantener " y clases fáciles-a-documento son mucho preferible a una clase sola grande y compleja con muchas responsabilidades (mirar a Dios el anti modelo de Clase encima). Una pelota enredada de hilo. Clases grandes tienen una tendencia de convertirse en el equivalente de software de esto**.**

Esté consciente de cuando clases adicionales pueden simplificar el diseño y decuple las partes innecesariamente acopladas de su código.

Examples and signs

Ejemplo clasico:

```
class Shape:
    def __init__ (self, shape_type, *args):
        self. shape_type = shape_type
        self. args = args

    def draw(self):
        if self. shape_type == "circle":
            center = self. args [0]
            radius = self. args [1]
            # Draw a circle...
        elif self. shape_type == "rectangle":
            pos = self. args [0]
            width = self. args [1]
            height = self. args [2]
            # Draw rectangle...
```

Now compare it with the following:

```
class Shape:
    def draw(self):
        raise NotImplemented ("Subclasses of Shape should implement method 'draw'.")


class Circle(Shape):
    def __init__ (self, center, radius):
        self. center = center
        self. radius = radius

    def draw(self):
        # Draw a circle...
```

```
class Rectangle(Shape):
    def __init__ (self, pos, width, height):
        self.pos = pos
        self. width = width
        self. height = height

    def draw(self):
        # Draw a rectangle...
```

The tricky part

La adición de clases no es una bala mágica. La simplificación del diseño por rompiendo clases grandes requiere el análisis pensativo de las responsabilidades y exigencias.

tl; dr

Más clases son no necesariamente un signo de diseño malo.