Carrera:

Ingeniería de Software

NRC: 13364

Asignatura:

Aseguramiento de la Calidad De Software

Nombre del profesor:

Roberto Omar Andrade Paredes

Estudiantes:

Kevin Daniel Castillo Beltrán

Josué Alejandro Moreno Herdoiza

Ednan Josué Merino Calderón

Fecha:

22/07/2023

1. INTRODUCCIÓN

Los anti-patrones, también conocidos como trampas, son ejemplos ampliamente documentados de soluciones inadecuadas para problemas. Se estudian con el propósito de evitarlos en el futuro y, en caso de que se presenten, para reconocer fácilmente su presencia al investigar sistemas disfuncionales durante una auditoría.

El término "anti-patrón" surge como una contraparte al concepto de "patrón," utilizado en la arquitectura de software para describir las mejores prácticas en programación, diseño o gestión de sistemas. Por lo tanto, un sistema considerado "bien hecho" estaría repleto de patrones y debería carecer de anti-patrones; idealmente, esto sería lo deseable.

2. ANTIPATRONES

Existen tres anti-patrones en general que serán especificados:

1. **De Codificación**
   - 1.1. **Lava Flow:** Construir grandes cantidades de código de manera caótica, con escasa documentación y poca claridad sobre su función en el sistema, genera lo que se conoce como flujos de lava. A medida que el sistema avanza en su desarrollo y crece, estos flujos de lava tienden a solidificarse, lo que implica que corregir los problemas que surgen se vuelve mucho más complicado, y el caos aumenta de manera exponencial.
   - 1.2. **The God:** Un programa omnipresente y enigmático es aquel sistema en el que una sola clase o módulo (como la función principal o equivalente) se encarga de todo. En consecuencia, el programa se convierte en un solitario archivo único con una gran cantidad de líneas. Esto resulta en un código desorganizado y altamente interdependiente, donde todas las funcionalidades se encuentran estrechamente entrelazadas.
   - 1.3. **Golden Hammer:** Este fenómeno, popularmente conocido como la "técnica de la barita mágica," es un vicio que implica aferrarse a un paradigma específico para solucionar todos los problemas que surgen durante el desarrollo de sistemas. Un ejemplo común es la tendencia a utilizar siempre el mismo lenguaje de programación, ya sea .NET, Java o PHP, sin considerar si es la elección más adecuada para cada desarrollo en particular. Es crucial tener en cuenta que cada uno de estos lenguajes posee capacidades y limitaciones que los hacen más apropiados

para ciertas aplicaciones. Este comportamiento se caracteriza por dos aspectos principales: en primer lugar, el uso obsesivo de una herramienta en particular, y, en segundo lugar, la terquedad de los desarrolladores al aplicar un único paradigma de solución en todos los programas. Esta actitud puede resultar en un consumo excesivo de esfuerzo para resolver un problema, ya que no siempre se adapta eficientemente a cada situación específica. Es importante reconocer que la diversidad de herramientas y enfoques puede llevar a soluciones más efectivas y eficientes en el desarrollo de sistemas.

1.4. **Spaghetti Code:** Carece de documentación y donde cualquier pequeño cambio provoca convulsiones en toda la estructura del sistema. En términos coloquiales, es como si el proceso de codificación fuera realizado con poca precisión y cuidado. A diferencia del estilo volcán, que se enfoca en la forma en que el sistema crece con la adición de módulos, aquí la crítica se dirige a la forma en que se escriben cada una de las líneas de código, desde la identación hasta el uso de los lenguajes de programación y su interacción. Este tipo de código caótico puede dificultar enormemente el mantenimiento y la comprensión del sistema en su conjunto.

2. De Arquitectura

2.1. **Reinventar la Rueda:** Este fenómeno se refiere a la práctica de re-implementar componentes que podrían obtenerse preconstruidos y hacer un uso limitado del reuso de código. En pocas palabras, se trata de querer construir todo desde cero. Esto conlleva a varios problemas:

2.1.1. Escaso reuso de código entre proyectos, lo que significa que cada nuevo proyecto comienza prácticamente desde cero.

2.1.2. Constante reescritura de fragmentos de código que ofrecen la misma funcionalidad, redundando en esfuerzos innecesarios.

2.1.3. Un consecuente gasto inútil de mano de obra y tiempo al reimplementar cosas que ya existen y están disponibles.

2.1.4. El software resultante se vuelve innecesariamente más complejo y denso debido a la duplicación de esfuerzos.

En resumen, esta práctica disminuye la eficiencia y aumenta los costos en el desarrollo de software, al no aprovechar las soluciones ya existentes y disponibles para construir nuevos proyectos.

2.2. **Stovepipe:** Este término se refiere a la creación de "islas automatizadas" dentro de una misma empresa, donde cada departamento establece su propio subdepartamento

de sistemas. Estas "islas" operan de manera independiente y a menudo en conflicto entre sí. Cada una desarrolla la parte del sistema que necesita para satisfacer sus requerimientos particulares, sin considerar el conjunto completo (falta de un plan o guía central). El resultado es una escasa o nula interoperabilidad entre las distintas áreas, lo que implica una falta de reuso y, por ende, un aumento de los costos. Sorprendentemente, esta práctica se está volviendo más común debido a la urgencia de los departamentos específicos dentro de una empresa más grande, que buscan acceder rápidamente a Tecnologías de la Información. En resumen, la creación de estas "islas automatizadas" puede llevar a problemas de incompatibilidad y gastos innecesarios, ya que cada departamento trabaja de forma aislada sin tener en cuenta la integración y colaboración con el resto de la empresa.

3. De Administración de Proyecto

3.1. **The Mythical Month Man:** Es una creencia errónea que sugiere que asignar más personal a un proyecto acelerará su tiempo de entrega. Esta idea suele surgir como una respuesta desesperada para intentar corregir retrasos en el proyecto. Sin embargo, paradójicamente, llega un punto en el que asignar más personal en realidad conduce a mayores retrasos en el proyecto. En lugar de acelerar la entrega, agregar más miembros al equipo puede ocasionar una serie de problemas, como una comunicación menos efectiva, aumento de conflictos, mayor complejidad en la coordinación y más dificultades para mantener un enfoque unificado. Todo esto, en conjunto, puede ralentizar el avance del proyecto, haciendo que el trabajo sea menos eficiente y dificultando la finalización en el tiempo esperado. Es importante entender que el éxito de un proyecto no se logra simplemente sumando más recursos, sino a través de una planificación adecuada, una gestión efectiva del equipo y una comprensión realista de las capacidades y limitaciones de los recursos disponibles.

3.2. **Project Miss-managemen:** El síndrome del jefe descoordinado se presenta cuando la jefa o el jefe a cargo no saben cómo coordinar adecuadamente el proyecto. En consecuencia, el proyecto se descuida y no se monitorea de manera efectiva. En las etapas iniciales, puede ser difícil detectar este problema, pero repentinamente, surge de golpe y suele tener un impacto significativo en la situación del proyecto. Este síndrome se manifiesta a través de retrasos en las fechas de entrega y/o áreas incompletas. La falta de coordinación y seguimiento adecuado provoca que las tareas no se realicen a tiempo o que queden pendientes, lo que afecta negativamente el

desarrollo del proyecto en su totalidad. Es esencial que los líderes del proyecto sean capaces de coordinar y supervisar de manera efectiva para evitar este tipo de situaciones. La buena gestión, la comunicación clara y el seguimiento regular son fundamentales para asegurar el éxito y evitar que el proyecto se vea afectado por el síndrome del jefe descoordinado.

3. PROCESO DE IDENTIFICACIÓN

A continuación, la serie de pasos para identificar Anti-Patrones de Software

- Paso 1: Conocimiento de patrones y buenas prácticas
- Paso 2: Revisión del código existente
- Paso 3: Análisis de problemas recurrentes
- Paso 4: Revisión de código en equipo
- Paso 5: Utilización de herramientas de análisis de código
- Paso 6: Investigación de casos de estudio
- Paso 7: Documentación y sensibilización

4. CONCLUSIONES

Algunos de estos patrones son tan irrisorios que podríamos pensar que nadie podría cometerlos, y, sin embargo, reflexionemos sobre cuántos de nosotros, ya sean estudiantes o profesionales, recurrimos a ellos como nuestra primera opción. Es importante mencionarlos para generar conciencia, pero cabe destacar que cada uno de estos anti-patrones merece un análisis profundo y minucioso de sus causas, síntomas, consecuencias y posibles soluciones. Conocer los anti-patrones resulta valioso para abordar procesos de refabricación, migración, actualización o reingeniería en los sistemas.

El avance vertiginoso de la tecnología informática es algo notable, y lo que eran consideradas mejores prácticas en una época, pueden convertirse en anti-patrones en el lustro siguiente. Por ejemplo, la programación estructurada, que en su momento fue una solución sencilla y rápida, hoy día puede llevar a consecuencias negativas a largo plazo en ciertos contextos.

Es esencial mantenerse actualizado y estar en constante aprendizaje para identificar y evitar estos anti-patrones en nuestros proyectos de desarrollo de software. La evolución de las tecnologías nos brinda nuevas oportunidades, pero también plantea nuevos desafíos que requieren un enfoque más cuidadoso y consciente para alcanzar soluciones óptimas y sostenibles en el tiempo.

5. BIBLIOGRAFÍA

5.1. J2EE AntiPatterns. Bill Dudney, Stephen Asbury, Joseph Krozak, and Kevin Wittkopf. Ed. John Wiley & Sons, 2003. ISBN: 0471146153.

5.2. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. William J. Brown. Ed. John Wiley & Sons, 1998. ISBN: 0471197130.

5.3. Antipatterns in Project Management. William J. Brown. Ed. John Wiley & Sons, 2000. ISBN: 0471363669