UNIVERSIDAD PRIVADA DE TACNA

FACULTAD DE INGENIERIA
Escuela Profesional de Ingeniería de Sistemas

Principios de Diseño y DDD

Curso: Calidad y Pruebas de Software

Docente: Mag. Ing. Patrick Cuadros Quiroga

Integrantes:

Valdez Apaza, Rafael Jesus (2019063317)
Poma Chura, Jhon Romario (2019064022)
Sagua Ramos, Gustavo Alonso (2018062254)
Romero Roque, Angelica Beatriz (2019063327)
Yucra Mamani, Vanessa (2019063635)
Cutipa Machaca, Arnold Felix (2019064040)

Tacna – Perú
2021

INDICE GENERAL

PRINCIPIOS DE DISEÑO Y DDD

I. Resumen

En el presente documento se hablará de un tema importante como son los principios de diseños, abordaremos los principios de SOLID, DRY, se hablará qué son y su importancia, también se dará una explicación básica de cada principio.

Domain-driven design es un enfoque para el desarrollo de software con necesidades complejas mediante una profunda conexión entre la implementación y los conceptos del modelo y núcleo del negocio.

El diseño guiado por el dominio no es una tecnología ni una metodología, este provee una estructura de prácticas y terminologías para tomar decisiones de diseño que enfoquen y aceleren el manejo de dominios complejos en los proyectos de software.

II. Abstract

In this document we will talk about an important topic such as design principles, we will address the principles of SOLID, DRY, what they are and their importance will be discussed, and a basic explanation of each principle will also be given.

Domain-driven design, is an approach to developing software with complex needs through a deep connection between the implementation and the concepts of the business model and core.

Domain-driven design is not a technology or a methodology, it provides a framework of practices and terminologies to make design decisions that focus and accelerate the management of complex domains in software projects.

The term was coined by Eric Evans in his book Domain-Driven Design - Tackling Complexity in the Heart of Software.

III. Introducción

Domain Driven Design (DDD) es una práctica de desarrollo de software que pone el acento en el Dominio del Negocio como faro del proyecto y en su Modelo como herramienta de comunicación entre negocio y tecnología. En el equipo de desarrollo de JPA empleamos Domain Driven Design como referencia para afrontar proyectos de desarrollo de cierta complejidad. Fruto de nuestros errores, pero, sobre todo de responder a las preguntas honestas de unos compañeros y los dardos envenenados de otros, hemos ido refinando su aplicación. En este artículo te presento algunas conclusiones y prácticas que pueden facilitarte la aproximación a sus principios.

Sus principios se basan en:

- Colocar los modelos y reglas de negocio de la organización, en el core de la aplicación
- Basar nuestro dominio complejo, en un modelo de software.
- Se utiliza para tener una mejor perspectiva a nivel de colaboración entre expertos del dominio y los desarrolladores, para concebir un software con los objetivos bien claros.

IV. Desarrollo

Principios SOLID

SOLID es el acrónimo que acuñó Michael Feathersque, engloba el nombre 5 principios, basándose en los principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000 en su paper “Design Principles and Design Patterns”.

Los 5 principios son:

- Single Responsibility

  El Single Responsibility Principle (SRP) o principio de responsabilidad única es un principio que defiende que una clase o módulo debería tener responsabilidad sobre una única funcionalidad del software. Este principio es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

  Beneficios del Principio de Responsabilidad Única:
  Los beneficios del principio de responsabilidad única son varios. A parte de brindarnos un controlador más limpio y legible, podemos enumerar los siguientes beneficios:
  - Cada clase se encarga de lo suyo.
  - Hay un lugar para todo y todo esta en su lugar.
  - Nombres más específicos para las clases.
  - Mayor facilidad para atacar el código con pruebas.

- Open/Closed

  Representa la O de SOLID y lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction”. Con este principio se pretende minimizar el efecto cascada que puede suceder cuando cambiamos el comportamiento de una clase. Si existen clientes que dependan de ella, es posible que tengan que cambiar su comportamiento también. Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías o frameworks.

  Beneficios del Principio de Abierto/Cerrado
  - Extender las funcionalidades del sistema, sin tocar el núcleo del sistema.
  - Prevenimos romper partes del sistema al añadir nueva funcionalidades.
  - Facilidad en el «testeo» de clases.
  - Separación de las diferentes lógicas.

- Liskov Substitution Principle

  La L de SOLID alude al apellido de quien lo creó, Barbara Liskov. El principio dice: una clase que hereda de otra debe poder usarse como su padre sin necesidad de conocer las diferencias entre ellas. Según Robert C. Martin incumplir el Liskov Substitution Principle (LSP) implica violar también el principio de Abierto/Cerrado.
  Beneficios de Principio de sustitución de Liskov:
  - Código más reusable.
  - Facilidad de entendimiento de las jerarquías de clase.
  - Aplicando este principio se puede validar que nuestras abstracciones estan correctas.

- Interface Segregation Principle
  La interface Segregation Principle (ISP) o principio de segregación de interfaces defiende que ninguna clase debería depender de métodos que no usa, es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.
  Beneficios del Principio de Segregación de la Interfaz:
  - Su principal función es brindarnos un código desacoplado, permitiéndonos mayor facilidad en la refactorización, modificación y despliegue de nuestro código.
- Dependency Inversion
  Representa la D de los principios S.O.L.I.D. El principio dice, los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones. El principio tiene como fin evitar depender de concreciones para minimizar el grado de acoplamiento entre los componentes.
  Beneficios del Principio de Inversión de dependencia
  - Podemos crear clases más pequeñas e independientes, permitiéndonos, además, reutilizar lógica de una forma muy sencilla
  - Nos proporciona flexibilidad y estabilidad a la hora de desarrollar nuestras aplicaciones. Ya que estaremos más seguros a la hora de ampliar funcionalidades, etc.

Importancia:

- Permiten crear software estructurado correctamente que resista el paso del tiempo.

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

- Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
- Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
- Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

Principio DRY

El principio DRY es un acrónimo del inglés “Don’t Repeat Yourself” que significa “No te repitas” y éste consiste en evitar las duplicaciones lógicas en el software.

Importancia:

- Código mantenible: evitar la repetición de código permite que, si alguna vez cambia la funcionalidad que estes repitiendo, no lo tengas que hacer en todos los lugares en los que lo repetiste.
- Reduce el tamaño del código: Esto lo hace más legible y entendible, porque hay menos código que entender. Los procesos para evitar la repetición implican nombrar el pedazo de código que estás reutilizando para identificarlo, esto hace el código más legible si lo nombraste bien.
- Ahorra tiempo: Al tener pedazos de código disponibles para reutilizarlos, en el futuro estás más preparado para lograr lo mismo en menos tiempo.

V. Conclusiones

En conclusión, Domain Driven Design es una técnica que implica dos transformaciones, del negocio al modelo y del modelo al software

El modelo juega un papel central para asegurar la correspondencia entre el software y el negocio al que debe representar. El modelo debe estar siempre presente en la mente, en los ordenadores y en las pizarras de los equipos multidisciplinares. Buscamos una mentalidad de "Policía del Modelo" que persiga cualquier desviación de la consistencia de esta correspondencia.

Este es el primero de una serie sobre lo que hemos aprendido aplicando DDD en Jerónimo Palacios y Asociados. En el próximo, bajaré a la transformación del modelo en diseño. Algunas soluciones propuestas por DDD que hemos empleado y otras que no.

VI. Recomendaciones

Esta técnica fue ideada para el desarrollo de aplicaciones complejas y está orientada a proyectos que utilicen metodologías agiles; uno de los aspectos más complicados de los proyectos de software complejos que están en el dominio del mundo real al que sirve el software y no en su implementación es en ese punto en el que el DDD te podrá ayudar a tener una mejor visión y enfoque para evolucionar a través de sucesivas iteraciones de diseño.

VII. Bibliografía

Eric Evans. Strategic Design – Making models work in large projects.

Eric Evans. Domain-Driven Design: Tackling complexity in the heart of Software.

Hunt y Thomas. Pragmatic Unit Testing

UPC - EPE - Ingeniería de Sistemas - Programa de Actualización Profesional

Martín, M. J. SOLID: los 5 principios que te ayudarán a desarrollar software de calidad.

Magaz, A. ¿Qué aporta Domain-Driven Design al software?