Patrones de arquitectura - Clase 1

Introducción

Los problemas del monolito:

- A medida que avanzan los sprints se reduce la productividad entonces es mejor repensar la calidad de la arquitectura.

Respecto al valor del cambio: Es mejor un programa *que no funcione* pero que sea fácil de cambiar a un programa perfecto que es *imposible de cambiar*. Esto dado que los requerimientos cambian todo el tiempo.

Respecto a los paradigmas: Antes se programaba de forma secuencial pensando en componentes y servicios, de forma modular. La POO, mediante el uso de polimorfismo, se puede tener un control absoluto sobre cada dependencia del código fuente en el sistema -> permite crear una arquitectura de plugins, entonces en el alto nivel tenés las interfaces (políticas) y los detalles a bajo nivel se delegan a los módulos concretos. En el caso de la programación funcional dice que los arquitectos harían bien en concentrar todo el procesamiento posible en los componentes inmutables y sacar todo el código posible de aquellos componentes que deben permitir la mutación lo que permitiría la escabilidad.

En resumen el aporte es

- Estructural -> transferencia de control directo (*ejecuta esto*)
- Objetos -> transferencia de control (*ejecuta esto pero dependiendo de la clase*) permite definir cierto protocolo y la ejecucción dependerá de la clase que se esta invocando
- Funcional -> manejo de la asignación de variables (cosas inmutables) para poder escalar los componentes.

Al pensar en arquitectura es escencial pensar en el diseño basado en los requerimientos funcionales y no funcionales. Respecto a los últimos se tienen en cuenta:

- Performance
- Seguridad
- Usabilidad
- Evolución
- Concurrencia
- Portabilidad

Entonces el equipo de AS

- Justificar a sponsors la necesidad de su existencia
- Definición de la arquitectura: relevar requerimientos (funcionales y no funcionales), relevar la vista de los stakeholders, relevar restricciones de hardware, de so, de capacitación, documentar teniendo en cuenta de los stakeholders.
    - Evaluar estilos arquitectónicos
    - Definir como mapear cada uno de los elementos del desarrollo en la arquitectura: como se comunican los diferentes componentes.
    - Evaluar patrones arquitectónicos

    - Definición conceptual de la arquitectura
- Proveer la plataforma de desarrollo: conjunto de frameworks, herramientas para *no reinventar la rueda*.
- Documentación: modelar la vista para cada uno de los stakeholders, además es conveniente incluir diagramas conceptuales de la arquitectura orientado a la vista que corresponda.
- Capacitación y soporte: a usuarios y desarrolladores.

Arquitectura

Solución conceptual + plataforma de desarrollo que permita al desarrollador identificar y organizar cada uno de los elementos necesarios para el desarrollo de un sistema, simplificando:

- El cumplimiento de los reqs. funcionales y no funcionales.
- Un diseño y una implementación elegante y simple para su posterior corrección y extensión.
- Un diseño y una implementación en pro de la mejora de productividad del equipo.

Principios SOLID:

- S - The Single Responsibility Principle: Funcinoes o clases tengan responsabilidad simple, el módulo atienda a un actor. A nivel de arquitectura se convierte en el eje de cambio responsable de los límites arquitectónicos. Que un solo cambio no afecte en el resto de aspectos.

- O - The Open-Closed Principle: para que los sistemas de sw sean fáciles de modificar deben diseñarse de modo que su comportamiento puede cambiarse añadiendo nuevo código en lugar de modificar el existente. Los arquiectos separan la funcionalidad a partir de cómo, por qué y cuándo se modifica, y luego organizan esa funcionalidad separada en una jerarquía de componentes.

- L - The Liskov Substitution Principle: para construir sistemas de software a partir de piezas intercambiables -> uso de herencias e interfaces para aprovechar el polimorfismo, que uno pueda suscribirse a una interfaz y después cada pieza use la implementación que se requiera. Al vincularse al contrato de una clase (su interfaz), la implementación específica puede ser sustituida siempre y cuando se mantenga ese contrato.

- I - The Interface Segregation Principle: se aconseja a los diseñadores de software evitar depender de cosas que no utilizan. Un ejemplo es depender de un framework que a su vez depende de una base de datos; cualquier cambio en la base de datos podría impactar el framework y, por ende, el sistema que lo utiliza. Otro ejemplo es una clase con varios métodos donde diferentes usuarios solo utilizan un subconjunto de esos métodos. Si se realiza un cambio en un método no utilizado por un usuario, este igualmente podría necesitar recompilar su código debido a la dependencia de la clase completa. La solución propuesta es que cada usuario dependa de interfaces más específicas que contengan solo los métodos que realmente necesitan. Esto minimiza las dependencias innecesarias y reduce el impacto de los cambios.

- D - The Dependency Inversion Principle: los sistemas más flexibles son aquellos en donde las dependencias del código se refieren a abstracciones y no a concreciones, es decir, no instanciar clases concretas directamente, no heredar de clases concretas, no sobreescribir funciones concretas, no mencionar el nombre de nada concreto y volátil. En su lugar se sufiere el uso de factories para crear objetos volátiles.

Anexo: términos clave

- Monolito: Una arquitectura de software donde todos los componentes de una aplicación están acoplados y desplegados como una única unidad.
- Sprint: Un periodo de tiempo corto y fijo en metodologías ágiles (como Scrum) durante el cual se trabaja para completar un conjunto específico de tareas.
- Productividad: La eficiencia con la que un equipo de desarrollo puede producir software funcional y de valor.
- Atributos de Calidad (Requerimientos No Funcionales): Características de un sistema de software que describen cómo funciona en lugar de qué hace, como escalabilidad, mantenibilidad, seguridad, rendimiento y usabilidad.
- Paradigma de Programación: Un estilo fundamental de programación de computadoras, cada uno con su propio conjunto de conceptos y enfoques (por ejemplo, estructurada, orientada a objetos, funcional).
- Programación Estructurada: Un paradigma que enfatiza el uso de estructuras de control secuenciales, de selección (condicionales) e iterativas (bucles), evitando el uso indiscriminado de saltos (como "goto").
- Modularización: El proceso de dividir un sistema de software en módulos o componentes más pequeños e independientes.
- Programación Orientada a Objetos (POO): Un paradigma que organiza el software alrededor de "objetos" que encapsulan datos (atributos) y comportamiento (métodos).
- Polimorfismo: La capacidad de objetos de diferentes clases de responder al mismo mensaje (o invocación de método) de manera específica a su tipo.
- Arquitectura de Plugins: Un diseño arquitectónico que permite extender la funcionalidad de una aplicación mediante la adición de componentes independientes (plugins) que se adhieren a una interfaz o protocolo definido.
- Programación Funcional: Un paradigma que trata la computación como la evaluación de funciones matemáticas y evita o minimiza los cambios de estado y los efectos secundarios.
- Inmutable: Un objeto o componente cuyo estado no puede ser modificado después de su creación. Stateless (Sin Estado): Un servicio o componente que no guarda información sobre las sesiones o interacciones previas con el cliente. Cada solicitud se trata de forma independiente.
- Transferencia de Control: El mecanismo por el cual la ejecución de un programa pasa de una parte del código a otra (por ejemplo, la invocación de una función o método).
- Seguridad: Las medidas para proteger un sistema de software contra accesos no autorizados, uso indebido o robo de datos.
- Disponibilidad: La capacidad de un sistema de software de estar operativo y accesible cuando se requiere.
- Integración: La capacidad de un sistema de software de interactuar y compartir datos con otros sistemas.
- Performance (Rendimiento): La capacidad de un sistema de software de responder a las solicitudes y realizar tareas de manera eficiente en términos de tiempo y recursos.
- Stakeholder: Cualquier persona o grupo que tiene un interés en un sistema de software, incluyendo usuarios finales, desarrolladores, clientes, gerentes y operadores.
- Framework: Una estructura conceptual y tecnológica de soporte, típicamente con componentes y herramientas preconstruidas, para el desarrollo de software.

- Documentación: La información escrita que describe la arquitectura, el diseño, la funcionalidad y el uso de un sistema de software.
- Patrón de Arquitectura: Una solución general y reutilizable a un problema común en el diseño de software dentro de un contexto dado.
- Cliente-Servidor: Un patrón donde un cliente realiza solicitudes a un servidor, que a su vez proporciona los recursos o servicios solicitados.
- Arquitectura en Capas: Un patrón que organiza un sistema en múltiples niveles o capas, donde cada capa tiene responsabilidades específicas y se comunica con las capas adyacentes.
- Autenticación: El proceso de verificar la identidad de un usuario o sistema.
- Autorización: El proceso de determinar qué acciones o recursos puede acceder un usuario o sistema autenticado.
- Dependencias: Las relaciones entre diferentes partes de un sistema de software, donde un componente necesita o utiliza otro componente.
- Refactorización: El proceso de reestructurar el código existente sin cambiar su comportamiento externo, con el objetivo de mejorar su legibilidad, mantenibilidad o rendimiento.
- Deadlock: Una situación en la que dos o más procesos están bloqueados indefinidamente, esperando recursos que están siendo retenidos por otros procesos.
- Single Responsibility Principle (SRP): Un principio de diseño que establece que una clase o módulo debe tener una sola razón para cambiar.
- Open/Closed Principle (OCP): Un principio de diseño que establece que una entidad de software (clase, módulo, función, etc.) debe estar abierta para la extensión pero cerrada para la modificación.
- Liskov Substitution Principle (LSP): Un principio de diseño que establece que los subtipos deben ser sustituibles por sus tipos base sin alterar la correctitud del programa.
- Interface Segregation Principle (ISP): Un principio de diseño que establece que ninguna cliente debe ser forzado a depender de interfaces que no utiliza.
- Dependency Inversion Principle (DIP): Un principio de diseño que establece que las entidades de alto nivel no deben depender de entidades de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.
- Factory: Un objeto que se encarga de la creación de otros objetos, abstrayendo el proceso de instanciación.
- Volátil: En el contexto del principio de inversión de dependencias, se refiere a clases concretas cuya implementación es propensa a cambios frecuentes.