Arquitectura de Software

Es una descripción de los subsistemas de un sistema software, sus propiedades y las relaciones entre ellos. Realizar esta descripción nos permite:

- Facilita la comprensión a cualquier miembro del equipo
- Permite que cada miembro pueda trabajar en los subsistemas de forma individual
- Prepara al sistema para su extensión

- Facilita la reutilización del sistema

Elementos de una arquitectura software

- **Componentes**: Elementos computacionales donde se realiza el trabajo
  - Grano grueso: Servidor web
  - Grano fino: Módulo
- **Conectores**: Comunican componentes
  - Explicitos: Invocación
  - Implicitos: Eventos
- **Configuración**: Disposición concreta de los componentes y conectores

Estilos de Arquitectura Software

Expresa la organización estructural para sistemas software. Una buena arquitectura sigue un estilo o patrón arquitectónico que asegura la integridad conceptual.

Basados en flujos de datos

Tuberías y filtros

Los datos se procesan incrementalmente conforme llegan. Pueden generar la salida antes de consumir toda la entrada.

- Ventajas: Sencillos de mantener, composicional. Reutilizable
- Inconvenientes: Poca interactividad con el usuario. Sin estado global

MapReduce

Se basa en la programación funcional. Utiliza las funciones de map y reduce para aumentar el paralelismo, trabajando con pares `(key, value)`

Basados en llamada y retorno

Funcional (Subrutinas,llamdas)

Se basa en la descomposición funcional del sistema. Las subrutinas corresponden a las tareas a realizar. Se combinan según su interfaz y el flujo de control. Las llamadas permiten conectar las distintas subrutinas

- **Ventajas**: Partes bien identificadas y sencillo de sustituir
- **Inconvenientes**: Dependencia entre subrutinas. Dificil de extender

Orientado a objetos (Objetos,Mensajes)

Se basa en la descomposición en objetos. Modelan las entidades reales y colaboran intercambiando mensajes. Encapsulan un estado privado y mantienen su integridad, exhibiendo un comportamiento público. Se conectan mediante mensajes

- **Ventajas**: Reducen las dependencias y se puede distribuir

- **Inconvenientes**: Los objetos deben de conocerse para cooperar. Existen efectos secundarios

Arquitectura de capas (Capas,Interfaces)

Se organizan en niveles de abstracción. Cada capa se comunica exclusivamente con las adyacentes utilizando unas interfaces bien definidas. Nos permiten un nivel de abstracción creciente conforme ascendemos por las capas

- **Ventajas**: Reutilizable y mantenible
- **Inconvenientes**: Menor rendimiento

Clientes servidor (Servidor-cliente,Flujo de datos)

Los datos y el procesamiento se distribuyen a lo largo de varios procesadores. Centraliza la gestión de la inforamción y se basan en un protocolo de petición y respuesta

- **Ventajas**: Centralizado, escalable
- **Inconvenientes**: Poco robusto, congestión del tráfico

Modelo vista-controlador (Modelo-vista-controlador, Interfaz)

Ayuda a separar la capa de interfaz de usuario de las otras partes del sistema. Es adecuando cuando existen aplicaciones con interfaces de usuario interactivas o muchas formas de interactuar con los datos

- **Ventajas**: Robusto, flexible
- **Inconvenientes**: Mayor sobrecarga

Elementos del MVC

- **Modelo**: Representa el problema a resolver y encapsula el estado de la aplicación. Contiene la lógica del negocio
- **Vista**: Contiene los controles interactivos que lanzan eventos, pero no los maneja
- **Controlador**: Responde a los eventos de la vista, indicando que acciones deben de ocurrir en el modelo y actualizando la vista

Diseño de software

Proceso creativo de transformar un problema en una solución. La descripción de una solución también se conoce como diseño. Se separa en dos fases:

- **Análisis**: Plantea el problema (qué)
- **Diseño**: Especifica una solución particular para el problema (como)

Tomamos una decisión en de diseño en base a los requisitos, la tecnología disponible y la expreriencia

ISO/IEC 25010 Product Quality Model

Se usa como piedra angular en la evaluación de la calidad de un producto software determinado. La calidad es el grado en el que el producto satisface los requisitos de los usuarios, aportando de esa forma un valor a la organización. Se compone de las siguientes características:

- **Adecuación funcional**: satisface las necesidades declaradas
- **Eficiencia de desempeño**: Cantidad de recursos utilizados bajo determinadas condiciones
- **Compatibilidad**: Capacidad para intercambiar información y realizar sus funciones compartiendo entorno
- **Usabilidad**: Capacidad del producto para ser entendido o aprendido
- **Fiabilidad**: Capacidad de realizar unas funciones especificadas bajo condiciones determinadas
- **Seguridad**: Protección de la información de manera que personas o sistemas no autorizados no puedan leerlos o modificarlos
- **Mantenibilidad**: capacidad del producto para ser modificado efectiva y eficientemente
- **Portabilidad**: Capacidad del producto o componente de ser transferido de forma eficiente a otro entorno

Principios de diseño

Modularidad

Divide y venceras. Dividimos el sistema en componentes según algún criterio

Aumentar la cohesión

Forma en la que agrupamos las unidades del software en una unidad mayor siguiendo un criterio. Dicha unidad será más sencilla de diseñar, implementar y mantener

Tipos de cohesión

- **Cohesión funcional**: Los elementos del módulo están relacionados en el desarrollo de una única función.
- **Cohesión secuencial**: Un módulo realiza distintas tareas en secuencia, de forma que las entradas de cada tarea son las salidas de la tarea anterior. No es una mala cohesión si las tareas implicadas no son muy complejas y requieren pocas líneas de código.
- **Cohesión comunicacional**: El módulo realiza actividades paralelas usando los mismos datos de entrada y salida
- **Cohesión procedimental**: El módulo tiene una serie de funciones relacionadas por un procedimiento efectuado por el código (a modo de biblioteca). Es similar a la secuencial, pero puede incluir el paso de controles. Será deseable que las funciones estén relacionadas o realicen tareas dentro del mismo ámbito (p.e. la biblioteca string.h de C contienen funciones para operar con cadenas de caracteres).
- **Cohesión temporal**: Los elementos del módulo están implicados en actividades relacionadas con el tiempo.
- **Cohesión lógica**: Las actividades que realiza el módulo tienen la misma categoría. Esto es, es como si se tuvieran partes independientes dentro del mismo módulo.
- **Cohesión casual o coincidente**: Los elementos del módulo contribuyen a las actividades relacionándose mutuamente de una manera poco significativa. Este tipo de cohesión viola el principio de independencia y de caja negra de los módulos.

Reducir el acoplamiento

Grado de relación de un módulo con los demás. Cuanto menor sea su relación, más sencillo será de diseñar, mantener, probar...

Grados de acoplamiento

- **Acoplamiento normal**: una unidad de software llama a otra de un nivel inferior y tan solo intercambian datos (por ejemplo: parámetros de entrada / salida).
- **Acoplamiento externo**: las unidades de software están ligadas a componentes externos, como por ejemplo dispositivos de entrada / salida, protocolos de comunicaciones, etc.
- **Acoplamiento común**: dos unidades de software acceden a un mismo recurso común, generalmente memoria compartida, una variable global o un fichero.
- **Acoplamiento de contenido**: ocurre cuando una unidad de software necesita acceder a una parte de otra unidad de software.

Principio SOLID

Resume los 5 principios básicos del diseño Orientado a Objetos

Single Responsibility Principle (SRP)

Una clase solo tiene un motivo para cambiar. Separa las responsabilidades conteniendo la propagación del cambio. Una clase debería tener un solo motivo para cambiar

Open-Closed Principle (OPC)

Adaptarse a nuevas situaciones sin modificar el código existente. Se puede conseguir mediante polimorfismo o vinculación dinámica. Los cambios deben extender pero no modificar lo que ya funciona. Las clases deben estar abiertas a la extensión y cerradas a la modificación

Liskov Substitution Principle (LSP)

Diseño por contrato y herencia (DBC): Las clases herederas tienen que respetar el contrato de la clase base. Las subclases deben poder sustituir las clases sin que el código del cliente lo note.

Interface Segregation Principle (ISP)

Segregamos todos los servicios en distintas interfaces, de tal forma que el cliente elige solo aquellas que necesita. Los clientes no deben depender de métodos que no utilizan

Dependency Inversion Principle (DIP)

Los módulos de alto nivel no deben depender de los módulos de menor nivel. Ambos deben depender de abstracciones. Las abstracciones no deben de depender de detalles.

Patrones de diseño

Los patrones de diseño plantean soluciones parciales a los problemas de diseño recurrentes. Son una solución probada que se puede aplicar a un determinad tipo de problemas que se repiten en el desarrollo de software. Se basan en la experiencia previa. Los lenguajes orientados a objetos facilitan la reutilización de código, pero un buen diseño es la clave para una reutilización efectiva. Cada patrón se compone de 4 elementos:

- **Nombre**:
- **Problema**: Contexto de uso
- **Solución**: Responsabilidades, elementos, relaciones
- **Consecuencias**: Resultados de aplicar el patrón

Se clasifican según:

- **Según su proposito**:
  - **Patrones creacionales**: Cuando y como crear instancias
  - **Patrones estructurales**: Como combinar objetos
  - **Patrones de comportamiento**: Como distribuir las responsabilidades
- **Según su ámbito**:
  - **Patrones de clases**: Tratan sobre las relaciones entre clases y sus subclases
  - **Patrones de objetos**. Dichas relaciones pueden cambiar en el tiempo de ejecución y son dinámicas

Patrones de creación/diseño

Unitario (singleton)

Usados para cuando necesitamos una única instancia de ellos

Fábrica abstracta

Se utiliza cuando necesitamos crear instancias de clases que aún no conocemos. Creamos las instancias a través de la interfaz, comportandose como un constructor virtual

Patrones estructurales

Método de la fachada

Utilizamos una clase de fachada que facilite una interfaz sencilla de alto nivel, unificando todas las interfaces del sistema

Patrones de comportamiento

Estrategia

Nos permite sustituir un comportamiento utilizando una interfaz. Usamos una clase de contexto que implementa dicha interfaz, permitiéndonos la interoperabilidad

Ventajas de los patrones de diseño

- Soluciones concretas a ciertos problemas
- Soluciones técnicas
- Se aplican en situaciones muy comunes
- Soluciones simples

Inconvenientes de los patrones de diseño

- Uso poco intuitivo
- Es difícil reutilizar dichas implementaciones

Refactorización

Son cambios realizados en la estructura interna de un producto software para facilitar su comprensión o futuros cambios. No modifican el comportamiento del software. La refactorización es importante tras añadir nuevas funciones al sistema o tras revisar el código. Son candidatos a ser refactorizados:

- Código duplicado
- Métodos voluminosos
- Clases muy grandes