Juan David Cerquera Salazar



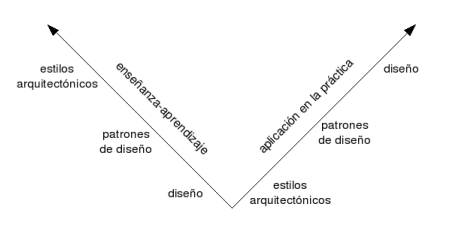
Soy Juan David Cerquera tengo 19 años, actualmente me encuentro cursando un tecnólogo de análisis y desarrollo software en el SENA y una ingeniería en la UNAD. Empecé a desarrollarme como programador desde finales de mis años escolares con proyectos básicos, lo que me ha permitido desarrollar gusto a la programación, aspiro a llegar ser un gran desarrollador FullStack.

Tengo alrededor de dos años de estudio en el ámbito de la programación y la tecnología, teniendo conocimientos en lenguajes de programación como java, c#, Python y JavaScript. También he tenido experiencia con base de datos MySQL, y con algunos framework como angular y spring boot.

**Introducción a la Arquitectura de Software**

En los primeros años de la construcción de software no existía el diseño del sistema como una etapa independiente la programación. Pues esta practica se comienza a proponer, investigar y aplicar a principios de los 70.

Mientras que en el 90 se empiezan a ver la necesidad de desarrollar un nivel de abstracción superior al de diseño, aquí es donde surge la arquitectura de software.



“Las etapas de la fase Arquitectura del Sistema se enseñan-aprenden de forma opuesta a la forma en que se aplican en la práctica”.

1. Elección del estilo arquitectónico

2. Selección de los patrones de diseño

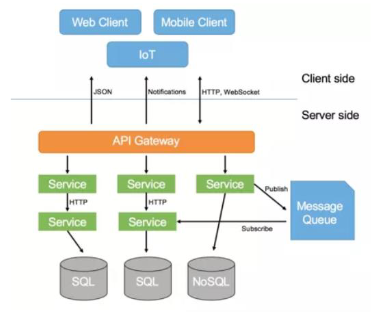
3. Diseño de componentes

Hay que tener en cuenta que jamás se contara con los requerimientos completos de un sistema, sino que para definir el diseño se tiene en cuenta los requerimientos que se tiene y los que llegaran.

A la arquitectura de software afecta a su entorno y su entorno la afecta a ella esto se denomina ABC (Architecture Business Cycle).

La arquitectura de software de un sistema es el resultado de combinar decisiones técnicas, sociales y del negocio. Pues los interesados del sistema se tarden o temprano requieran que tenga distintas características; Los requerimientos funcionales y no funcionales se deben tener en cuenta desde el principio ya que luego serán un gran problema.

**Arquitectura basada en micro-servicios para aplicaciones web**

****

Tomando como referencia las arquitecturas que tienen grandes empresas como Netflix y Amazon basándose en la arquitectura de microservicios. Nos habla de que la arquitectura de microservicios se trata de pequeños servicios autónomos, pequeños y con su única implementación individual de funcionalidad y También llegando a tener distintos host o servicios interactuando con protocolos http mediante APIs RESTFull.

Características:

* La descomposición de varias de sus partes funcionales de forma independiente lo que hace que implementar algún requerimiento evitando el redespliegue de toda la aplicación.
* Centrado en características y funcionalidades de forma independiente

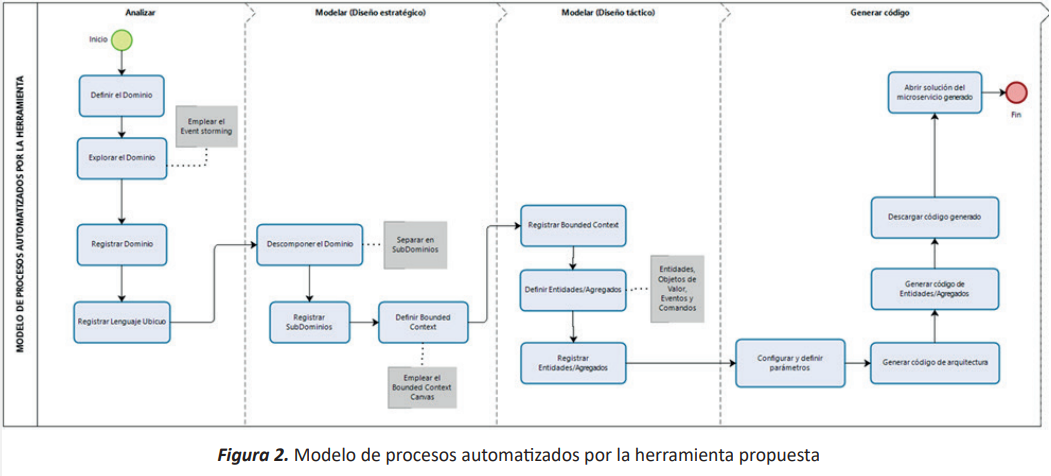
Ventajas.

* Esto nos da distintas ventajas como modularidad en las funcionalidades.
* Manejo independiente de despliegue de servicios

Desventajas.

* Alto consumo de memoria
* Tiempo para fragmentar distintos microservicios
* Necesidad de desarrolladores para la solución de problemas como latencia en la red o balanceo de cargas.

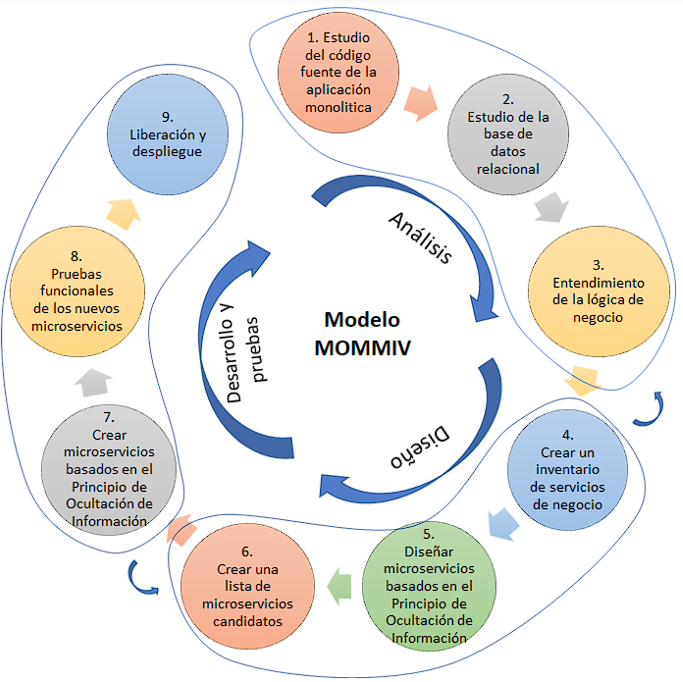
**Herramienta para el modelado y generación de código de Arquitecturas de Software basadas en Microservicios y Diseño guiado por el dominio (DDD)**

****

Desarrollo de una herramienta de modelado y generador de código de arquitectura de software basada en Microservicios y Diseño Dirigido por Dominio (DDD) que permita facilitar y acelerar el desarrollo de proyectos de software y la migración de sistemas heredados a nuevos servicios independientes, implementando nuevas tecnologías que permitirán al software tener interoperabilidad, modularidad, seguridad y escalabilidad.

Realizando pruebas con distintos arquitectos y desarrolladores de software expertos, donde se les dio instrucciones de cómo manejar la herramienta con el fin de implementarla en sus requerimientos individuales.

**MOMMIV: Modelo para descomposición de una arquitectura monolítica hacia una arquitectura de microservicios bajo el Principio de Ocultación de Información**



Se analizo que las empresas tienen softwares deficientes con el Patrón Arquitectónico Monolítico, el cual merma su funcionalidad con el paso del tiempo, estas necesitan una actualización, en este documento nos propone un modelo para migrar del patrón monolítico al patrón de microservicios. Siendo el punto de partida que no encontraron un modelo de migración usando el principio de Ocultación de información han creado el modelo MOMMIV (Modelo de Migración a Microservicios Versátil). Este estando dividido en Análisis, Diseño, Desarrollo.

Análisis: En esta etapa se analiza tanto el código fuente, componentes, dependencias y la base de datos relacional para tener la funcionalidad de la aplicación monolítica.

Diseño: Una vez analizado el software se procede a hace un “inventario” de las funcionalidades del programa, y se procede diseñar nuevos microservicios guiado por el principio de ocultación de información.

Desarrollo: se crean los microservicios basándose en el Principio de Ocultación de Información junto con su mecanismo de comunicación, Se realizan pruebas funcionales comparándolas con las funciones de la aplicación monolítica. Finalmente se procede a realizar el debido despliegue.

**Buenas prácticas en la construcción de software**

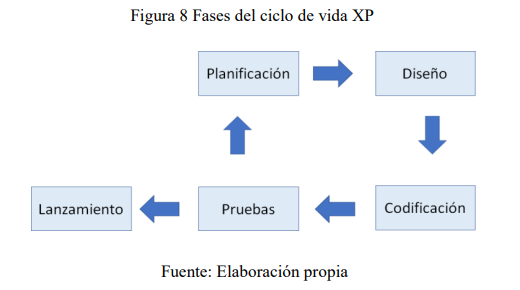
**Diagrama, Dibujo de ingeniería

Descripción generada automáticamente**Nos habla de las buenas prácticas que debemos seguir al desarrollar un software, dándonos a conocer de malas prácticas que debemos evitar como no tener una arquitectura diseñada para realizar un software de calidad, la falta de comunicación entre los miembros de un equipo lo cual afecta negativamente los estándares de calidad del software, llevando a la práctica de los anti-patrones.

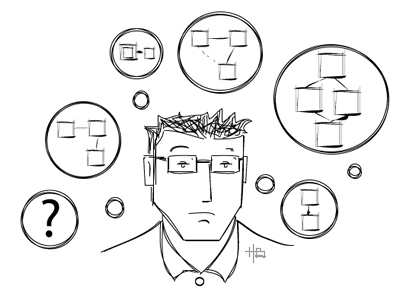
* copy and paste programming
* reinventing the Wheel

Buenas prácticas:

1. estándares y buenas prácticas: Aplicar estándares y buenas prácticas al momento de diseñar y escribir el código de las aplicaciones.
2. Clean Code, KISS: Mantener la simplicidad del código con el fin de facilitar la lectura de este.
3. Nombre con sentido: El nombre debe dar a entender su funcionalidad, siendo aconsejable mantener nombres cortos y evitar abreviaciones, prefijos y palabras redundantes y evitar usar palabras reservadas.
4. Funciones: Las funciones no deben ser muy complejas, siguiendo el principio de responsabilidad única y seguir la regla “the Stepdown rule”, “Ésta busca que cada función esté seguida de otra que corresponda con el siguiente nivel de abstracción”, en caso de usar parámetros evitar enviar más de tres y evitar repetir código.
5. Comentarios: Se deben usar cuando el código se da a entender por sí solo. Se recomienda usar en estos casos:
   1. Comentarios legales sobre derechos de autor.
   2. Comentarios informativos cuando no es clara una funcionalidad.
   3. Comentarios sobre advertencias.
   4. Comentarios sobre utilización de librerías o recursos de terceros que no sean modificables
6. Formato:
   1. Un archivo de código no debe sobrepasar las 500 líneas y cada línea no debe sobrepasar 120 caracteres.
   2. las variables deben ser declaradas lo más cerca de su implementación o en caso de ser usadas en todo el archivo se debe declarar en la parte superior.
   3. Los distintos desarrolladores deben tener el mismo estilo de indentación y formato.
7. Tratamiento de errores: Mantener un buen manejo de errores trata de preparar el código para cualquier eventualidad proporcionando suficiente información y mantener fácil de localizar dichos errores.
8. Clases y objetos: Tener clases que sigan el “principio de responsabilidad” única para evitar clases muy grandes y complejas, tener en cuenta el principio de “abierto/cerrado” introduciendo nuevas clases para evitar hacer una modificación de las existentes, seguir el principio de “inversión de dependencias” para facilitar las pruebas y tener un código más limpio.
   1. Acoplamiento: Tener un acoplamiento bajo (nos dice que las clases relacionadas no debe conocer muchos detalles de otra).
   2. Cohesión: Es la independencia de cada clase.
9. Arquitectura: Diseñar el sistema que cubrirá los requerimientos funcionales y no funcionales del software.
   1. Arquitectura Empresarial: Definir correctamente los componentes y activos de una empresa para entender la organización.
   2. Arquitectura Solución: Definir el diseño y comunicaciones de alto nivel del software para guiar el diseño y desarrollo a satisfacer las necesidades.
   3. Arquitectura Software: “Definir patrones de arquitectura, patrones de diseño, estilo arquitectónico, datos y tecnologías que se van a implementar en las soluciones.”
10. Estilos Arquitectónicos: Antes de decidir cual estilo de arquitectura escoger se debe analizar las características de cada uno de estos.
    1. Capas: Este divide la aplicación en capas cada una de estas con un rol definido por ejemplo presentación, servicios etc.
    2. Monolítico: Aplicación autosuficiente, agrupando todo en el mismo proyecto.
    3. Microservicios: Dividir el software en pequeños componentes con una única responsabilidad Permitiendo al software ser independiente de cada una de sus partes.
    4. Event-driven architecture: Esta arquitectura no espera respuestas inmediatas, sino que espera a un evento y reacciona a una respuesta.
    5. Cliente Servidor: Un servidor que brinda una serie de servicios o recursos los cuales son consumidos por el cliente.
11. Patrones de Diseño de software: Soluciones a errores de diseño, siendo generales y aplicables en casos específicos.
    1. Patrones creacionales: “Estos patrones intentan controlar la forma en que los objetos son creados, implementando mecanismos que eviten la creación directa de objetos.”
    2. Patrones estructurales: Estos patrones se encargan de diseñar la comunicación que tienen las clases.
    3. Patrones de comportamiento: “Son patrones que están relacionados con procedimientos y con la asignación de responsabilidad a los objetos.”
12. Herramientas: Antes de construir un software es recomendable realizar una investigación sobre las distintas herramientas disponibles.
13. Metodología XP: Mantener la comunicación con el cliente para realizar un software ajustado.
    1. Cliente: Proporciona los requerimientos y las principales necesidades.
    2. Desarrolladores: Aportar en la construcción y objetivos planteados.
    3. Testers: se encarga de comunicarse con el cliente y definir los estándares de calidad.
    4. Tracker: Realiza el seguimiento de las tareas propuestas y la comunicación con el cliente.
    5. Coach: Orienta al equipo de trabajo y cliente para que todo se haga de manera correcta.
    6. Manager: “El mánager se encarga de coordinar la comunicación entre distintas partes, gestionando los recursos necesarios.”
14. Ciclo de vida metodología XP:
    1. Fase de planificación: Entendimiento de los requisitos del cliente, con cada interacción generar una versión del software nueva y funcional.
    2. Fase de diseño: “En esta fase se manejan versiones sencillas haciendo lo mínimo necesario para que funcione y se obtenga un prototipo.”
    3. Fase de codificación: Desarrollado principalmente en pares para asegurar el entendimiento del software.
    4. Fase de pruebas: Realización de pruebas automáticas constantemente.
    5. Fase de lanzamiento: “En esta fase se entrega el producto final al cliente, después de ser validado y probado en cada una de las historias de usuario planteadas, teniendo un software útil e incorporable al producto.”



**Lenguajes de Patrones de Arquitectura de Software: Una Aproximación Al Estado del Arte**



Nos habla de la evolución y de los estándares de software a través de los años incluyendo y haciendo énfasis en la arquitectura.

Años Cincuenta: Se tomaba muy en cuenta todas las ciencias relacionadas con el software y el hardware como las matemáticas, ciencias de la computación y demás.

Años Sesenta: Era informal y sin planificación a profundidad.

Años Setenta: Se introdujeron estándares de calidad al software, el clico de vida del software y se identificaron principios de diseño.

Años Ochenta: Busca las buenas prácticas, productividad y escalabilidad teniendo en cuenta las principales cosas beneficiosas para el software y el equipo de desarrollo.

Años Noventa: El software tomo gran fuerza en el área laboral teniendo grandes avances en el desarrollo de patrones de diseño y estandarización de procesos.

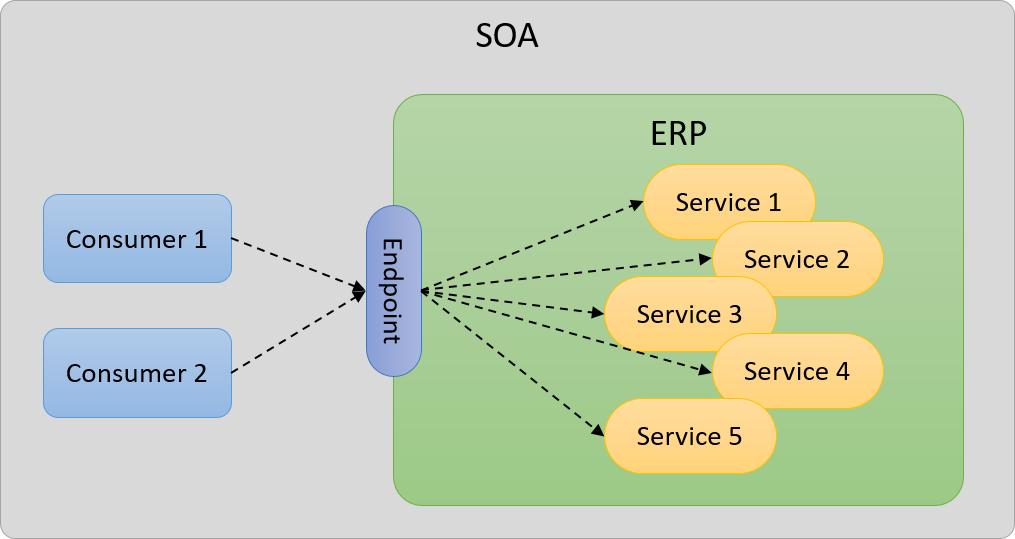
Ingeniería de Software: Se preocupa por los métodos y principios de diseño necesarios para el proceso de construcción del software.

Arquitectura de Software: Se centra en el cómo construir el sistema, asegurando que los atributos de calidad del software sean adecuados y que cumplan con las expectativas del usuario final.

Los Patrones de Diseño y los Lenguajes de Patrones: Estos fueron una adaptación de la propuesta del arquitecto Christopher Alexander, que en un principio descompuso problemas complejos en problemas pequeños y daba soluciones generales que funcionaban en distintas situaciones y a estas soluciones le llamo patrones, luego este concepto se adaptó al software.

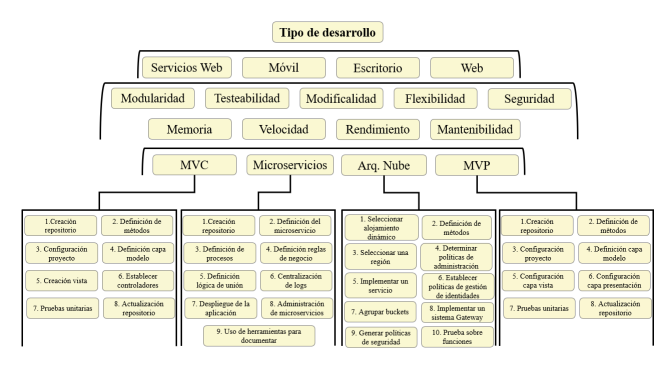
Lenguajes de patrones: Son agrupaciones de patrones de diseño para resolver problemas complejos y recurrentes

**ARQUITECTURA DE SOFTWARE, ESQUEMAS Y SERVICIOS**



Cada vez son más los requisitos empresariales por los que pasa el desarrollo de software teniendo tiempos de desarrollo cada vez mas cortos. La reutilización y el bajo acoplamiento son fundamentales para mejorar el desarrollo de software. La arquitectura de microservicios permite adaptarse a este modelo empresarial por su modularidad, su tiempo de desarrollo, su adaptabilidad, flexibilidad y tiempo de mantenimiento siendo un sistema flexible a los cambios del entorno.

**Marco de Trabajo para Seleccionar un Patrón Arquitectónico en el Desarrollo de Software**

**** **Clean architecture para mejorarG**

Se realizo un estudio a desarrolladores y arquitectos de software profesionales sobre cuáles son las arquitecturas de software más usadas.

Luego de analizar las distintas arquitecturas, se define cuáles son las más usadas son las más usadas, en que se destaca cada una.

Siendo las más usadas  
Arquitectura en la nube: Seguridad y flexibilidad.

MVC: Mantenibilidad, rendimiento, velocidad y memoria.

Microservicios: Mantenibilidad, Rendimiento, seguridad y flexibilidad.

MVP: Modificabilidad, Rendimiento, testeabilidad, flexibilidad, Modularidad.

También se analizó los distintos dispositivos a los que estaba enfocado esas arquitecturas:

Arquitectura en la nube: Aplicaciones web

MVC: Dispositivos móviles, Aplicaciones de escritorio, Aplicaciones basadas en la web

Microservicios: Aplicaciones web

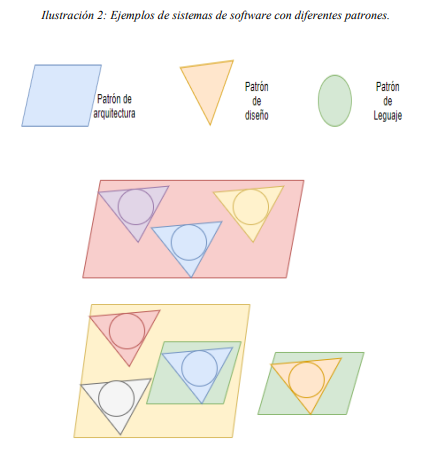
MVP: Dispositivos móviles y Aplicaciones web

Con estos y otras investigaciones realizadas por el equipo de trabajo se llego a desarrollar un modelo para escoger la arquitectura adecuada según sus requisitos.

Unos pasos generales del modelo son:

1. Selecciona el medio al que esta destinado el software (Móvil, WEB, Escritorio, Servicios Web)
2. El usuario escoge las 3 principales características necesarias en su software (Modularidad, Testeabilidad, flexibilidad, seguridad, modificabilidad)
3. Una vez escoja sus opciones le dará una recomendación de que arquitectura escoger.
4. También le dará la opción de ver como aplicar, esta le dará pasos que incluyen buenas prácticas y como se podría desarrollar ese patrón sin especificar lenguajes o tecnologías

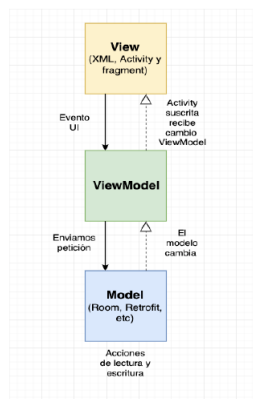
**Especificando una arquitectura de software**

****

Este articulo busca contextualizar la arquitectura de software dejando claro algunos conceptos relacionados a este.

1. ARQUITECTURA: las arquitecturas son un proceso creativo y estas van muy ligadas a lo que los arquitectos involucrados consideren convenientes basados en las siguientes tres fuentes:
   1. Método: Puede ser visto como una manera concisa y documentada, mediante el cual la arquitectura es derivada desde los requerimientos del sistema y las restricciones tecnológicas.
   2. Intuición: Habilidad de concebir sin razonamiento consciente.
   3. Reutilización: La mayoría de los elementos de una arquitectura son adoptados de otras arquitecturas, ya que el arquitecto puede estar familiarizado con la problemática o con alguna arquitectura encontrada en la literatura técnica.
2. ELECCIÓN DE PATRONES DE REFERENCIA: es importante entender que los patrones se dividen en varios tipos, los principalmente son:
   1. Patrones de arquitectura: Afectan la estructura global del sistema
   2. Patrones de diseño: Afecta a subsistemas o componentes del sistema global y sus relaciones.
   3. Patrones de lenguajes: describen como implementar ciertos aspectos de un problema utilizando las características específicas de dicho lenguaje.
3. Los patrones arquitectónicos se dividen en cuatro categorías
   1. Del lodazal a la estructura: Se centra en la organización de componentes desorganizados en una estructura organizada y modular. Siendo especialmente útil en sistemas complejos.
   2. Sistemas distribuidos: Se centra en la comunicación que tiene el sistema
   3. Sistemas interactivos: Se centra en mejorar la usabilidad de la aplicación separando la interfaz de usuario del núcleo de la lógica.
   4. Sistemas adaptables: Se centra en la adaptabilidad y la escalabilidad de un sistema
4. DEFINICIÓN DE COMPONENTES: Son los elementos que hacen parte de la arquitectura de software. Teniendo encueta como se relacionan.

**Análisis comparativo de Patrones de Diseño de Software**

****

artículo se detallan la estructura, componentes, ventajas y desventajas de los patrones de diseño.

1. Patrón Template Method - Modelo Plantilla: Es un patrón de diseño de comportamiento donde se define el esqueleto de un algoritmo en la superclase, mientras que, las subclases pueden sobrescribir los pasos del algoritmo sin la necesidad de cambiar su estructura.
   1. Ventajas
      1. Evita duplicación de código
      2. Es fácil de entender e implementar
      3. Flexible
   2. Desventajas
      1. Seguir el flujo es complejo a la hora de depurar
      2. Poca mantenibilidad
2. Model-View-Controller MVC: Separado en tres componentes Modelo, vista, controlador.
   1. Estructura:
      1. Modelo: Se encarga de manipular y gestionar los datos.
      2. Vista: Se encarga de mostrar las pantallas al usuario.
      3. Controlador: Se encarga de procesar las instrucciones recibidas y comunicar a la vista y al modelo.
   2. Ventajas:
      1. Modularidad.
      2. Mantenibilidad.
   3. Desventajas:
      1. Gran crecimiento del programa
      2. Implementación complicada con lenguajes que no sean compatibles con el paradigma orientado a objetos.
3. Model-View-Presenter MVP: Separado en tres componentes Modelo, vista, Presentador.
   1. Estructura:
      1. Modelo: Contiene los datos y la lógica, oculta su implementación con interfaces.
      2. Vista: Se encarga de mostrar las pantallas al usuario y recibir sus peticiones.
      3. Presentador: Recibe las peticiones de la vista e invoca métodos del modelo, obteniendo un resultado y actualiza la vista.
   2. Ventajas:
      1. Flexibilidad
      2. Permite la integración de distintas tecnologías
      3. La vista y el modelo pueden ser testeados de manera independiente.
   3. Desventajas:
      1. Implementación Compleja.
      2. No es adecuado para soluciones simples y pequeñas
4. Model Front Controller: Se centra en el manejo de peticiones, usando como punto inicial un controlador que realiza la gestión de solicitudes.
   1. Estructura:
      1. Controller: Procesa todos y gestionan las peticiones del sistema.
      2. Dispatcher: Se encarga de mostrar las pantallas al usuario y navegar entre vistas.
      3. Helper: ayuda al controlador y a la vista a terminar su procesamiento.
   2. Ventajas
      1. Existe una mejora significativa en la manejabilidad de la seguridad.
      2. Es posible reutilizar código.
      3. Evita tener distribuida la gestión de peticiones.
      4. Control centralizado
      5. Seguridad de subprocesos
   3. Desventajas
      1. Poca escalabilidad.
      2. Baja velocidad de respuesta
5. Model-View-ViewModel MVVM: permite aislar limpiamente la lógica de negocios y presentación de una aplicación de su interfaz de usuario.
   1. Estructura:
      1. Modelo: Es el responsable del acceso a la fuente de datos y de trabajar con esos datos.
      2. Vista: Se encarga de mostrar las pantallas al usuario y recibir sus peticiones.
      3. Vista del modelo: Encargada me transformar los datos en un formato que la vista lo requiera.
   2. Ventajas
      1. Facilita el mantenimiento
      2. Disminuye la cantidad de código
      3. Facilidad en pruebas unitarias
   3. Desventajas:
      1. Curva de aprendizaje alta
      2. Depuración compleja