INVETIGACIÓN DE REQUISITOS NO FUNCIONALES APLICADOS A PATRONES ARQUITECTÓNICOS

Trabajo 2

Andrés Elías Carrascal Verona, Luis Carlos Rendon Cardona, Wulfram Polo Casteñeda, Carlos Orrego Zapata

Universidad de Antioquia

Introducción

Mediante el siguiente trabajo, vamos a conocer acerca de los requisitos no funcionales aplicados a patrones arquitectónicos, conoceremos como están compuestos cada uno y cuál es el patrón arquitectónico indicado para estos RNF que día a día utilizan los desarrolladores para llevar a cabo una buena creación de software. Hablaremos de las características principales de estos RNF y su vinculación con los patrones arquitectónicos más utilizados. Brindando así una información en especie de guía para profundizar y afianzar los conocimientos previos.

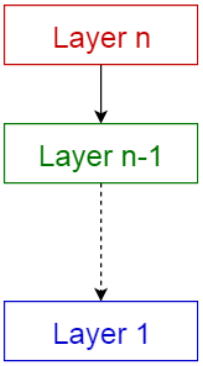
Objetivos generales

- Profundizar sobre los diferentes tipos de RNF que podemos utilizar a la hora de desarrollar software
- Investigar los distintos tipos de arquitecturas de software que podemos utilizar en los proyectos donde implementamos RNF
- Analizar la importancia de cada una de estas arquitecturas para el desarrollo de software.
- Ahondar en la práctica de la implementación de las arquitecturas de software
- Trazar diferencias entre los RNF y los RF
- Profundizar sobre cuando son aplicables los patrones de diseño a un requisito no funcional y cuando no

Objetivos específico

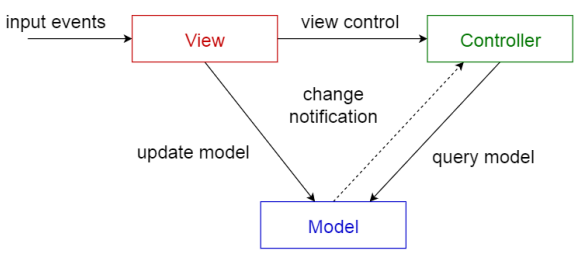
- Profundizar sobre las ventajas y desventajas de utilizar el patrón de diseño MVC
- Encontrar la similitud entre el patrón MVC Y el patrón de diseño por capas
- Ahondar sobre la investigación del patrón maestro-esclavo
- Investigar sobre cuando es mejor utilizar el patrón de diseño por capas

REGISTRO



El sistema deberá tener un archivo de log que contenga todas las transacciones realizadas en sistema, es decir un archivo en el que se conste cronológicamente cada uno de los acontecimientos que han tenido relevancia en el sistema y pueden ayudar a encontrar fallos, así como obtener rápidamente diversos informes sobre una determinada actividad. Por ende, el patrón de capas es el indicado para la funcionalidad de estos logs, ya que este patrón se utiliza para estructurar un sistema de tal forma que se pueda descomponer en diferentes capas en la cuales una capa le proporciona información a una capa superior teniendo así un registro de cada una de las transiciones del sistema desde las capas de presentación hasta las de acceso de datos, en donde cada log tiene una forma de identificarse dentro de cada capa para luego ser emitido entre capa y capa.

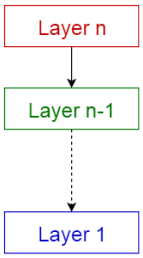
MANTENIBILIDAD



Podemos decir que la mantenibilidad como tal es una propiedad del software o del mismo desarrollo que se deriva del trabajo total que se necesita en determinadas circunstancias para que lo que sea que estemos programando se mantenga con un correcto funcionamiento luego de haber realizado algún cambio que se realice sobre el mismo. Debido a la necesidad de evolución, corrección o mejora, esta característica representa la capacidad de modificar los productos de software de manera efectiva y eficiente, además representa también la propiedad de hacer que un determinado software sea más llevadero y trabajable, permitiendo por ejemplo los cambios con impactos bajos en los demás softwares. Esta función se puede subdividir en las siguientes subcaracterísticas: Modularidad, reusabilidad, analizabilidad y capacidad para ser modificado.

Consideramos que la arquitectura más apropiada para aplicar la mantenibilidad del código es la del Modelo Vista Controlador, porque es aquella arquitectura que nos facilita más la vida en el momento de hacer una buena organización del código, siendo la medida más apropiada para implementarlo el hecho de dividir en varios paquetes cada funcionalidad en específico en la que estemos trabajando habiendo hecho primero una definición de todos los componentes con los que se trabajaría el programa para facilitar la división y unificar o integrar todo en el momento de la ejecución mediante el main o la clase principal.

MANEJO DE ERRORES

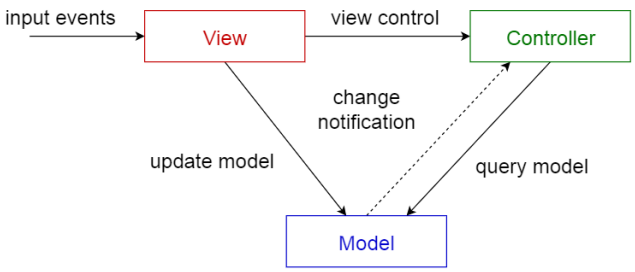


Uno de los factores que pueden llegar a afectar gravemente los programas o software de desarrollo que implementamos día a día es básicamente el manejo de errores, lo que nos lleva a intuir que un determinado programa que tenga un tratamiento efectivo de excepciones puede llegar a ser más óptimo en el momento de ejecución del mismo y nos puede facilitar en gran medida los problemas que se nos pueden generar debido al desconocimiento del manejo del software por parte del usuario en el momento de estar ejecutando la aplicación, es por eso que se debe implementar una propiedad al programa que proteja globalmente el código de estos errores que puede cometer el usuario y que puedan causar conflictos.

Concluimos entonces que la arquitectura más factible para trabajar el manejo de errores es el patrón de capas, utilizando un método en donde realizamos una división de la aplicación con la que estemos trabajando en un total de tres capas, esta técnica de manejo de aplicaciones por capas es conocida como three-tier applications, cuya característica fundamental es el hecho de separar la capa de lógica de negocios por ejemplo de la de entrada para la información y la capa de presentación, para entender mejor esta técnica lo explicaremos a través de los siguientes pasos:

1. Se realiza un encapsulamiento de la excepción
2. Hacemos un correcto almacenamiento de las características de dicha excepción
3. Es muy probable que la excepción venga de una capa anterior, lo mejor en estos casos es trasladar todas y cada una de las excepciones hasta la capa superior sin grabar previamente la primera, pues el proceso se debió hacer en la capa de origen.

INTERFACES

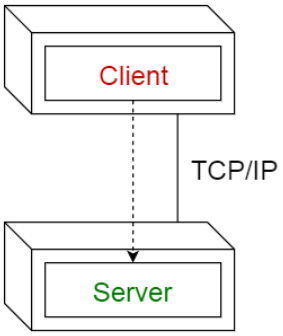


Cuando hablamos de interfaz apunta a la cara observable de los programas, como se presenta a los usuarios para que interactúen con un sistema. La interfaz gráfica involucra la presencia de una pantalla o monitor de ordenador constituida por una serie de menús e iconos que representan las opciones que el usuario puede tomar dentro del sistema.

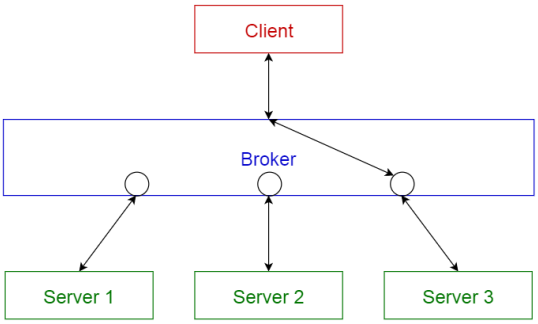
Este tipo de requisito describe la apariencia del producto. Es importante destacar que no se trata del diseño de la interfaz en detalle, sino que especifican cómo se pretende que sea la interfaz externa del producto. También pueden ser necesidades de cumplir con normas estándares, o con los estándares de la empresa para la cual se esté desarrollando el software.

Se puede implementar por mucho en el MVC (modelo vista controlador), este patrón como su nombre lo dice David la aplicación en tres partes, modelo, vista, controlador, la que nos interesa de estas tres capas es la Vista: dado que tenemos una capa especial para dedicarle al usuario intentando cumplir con los requisitos de intuitivismo combinados con los modelos especificados, sea de empresa, o cualquier otra índole.

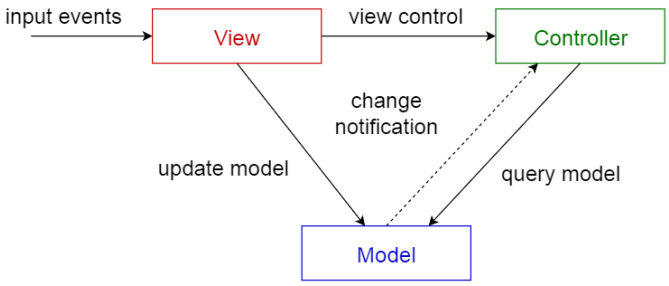
DEPENDIBILIDAD



El sistema debe tener una disponibilidad del 99,99% de las veces en que un usuario intente accederlo. El tiempo para iniciar o reiniciar el sistema no podrá ser mayor a 5 minutos. La tasa de tiempos de falla del sistema no podrá ser mayor al 0,5% del tiempo de operación total. El promedio de duración de fallas no podrá ser mayor a 15 minutos. La probabilidad de falla del Sistema no podrá ser mayor a 0,05. Por un lado, tenemos el patrón cliete-servidor, este patrón nos dice que se divide en dos partes; un servidor y múltiples clientes, El componente del servidor proporcionará servicios a múltiples componentes del cliente, con esto estaríamos cubriendo los múltiples componentes que conlleva el requisito no funcional. aunque podemos ver que estamos literal mente dependiendo del servidor, con estos nos referimos a la respuesta en caídas de este, fallos y otras falencias posibles. Por otro lado, tenemos el Patrón de intermediario Este patrón es utilizado para construir sistema distribuidos con partes desacopladas. Estas partes pueden interactuar entre si mediante servicios remotos. Una parte del intermediario es la responsable de la coordinación de la comunicación entre las partes. teniendo las tareas divididas podemos tener respuestas muchos más cortas en tiempo.

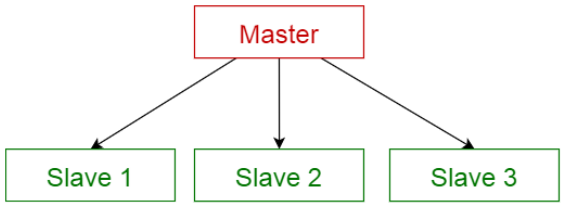


ACCESIBILIDAD



Cuando se habla de accesibilidad, se habla del grado por medio de la cual una personas puede usar un servicio u objeto; Cuando se realiza un software, normalmente los desarrolladores suelen ver el programa desde su punto de vista y no desde el punto de vista de un cliente, es aquí donde se cometen los errores de accesibilidad para el software, a la hora de realizar un software, se tiene que tener en cuenta a qué público va dirigido para así realizar un software muy accesible, algunos ejemplo serían: Que se accesible para personas con discapacidades motoras, que tenga problemas auditivos, personas de la tercera edad, entro otros. El patrón arquitectónico donde se puede implementar este requisito no funciona, es el patrón Modelo-vista-controlador (MVC), puesto que es aquí la aplicación se divide en 3 partes diferentes que son el modelo, la vista y el controlador, con ésta división, para el desarrollador es muy fácil centrarse en cómo debe quedar la ventana para que sea accesible por el usuario (Vista), una forma de cómo controlar los distintos end-point de cada petición con una estructura de documentación robusta (Controlador) y una forma de controlar cada objeto que se desea usar dentro de la aplicación; Es así, que la vista queda totalmente aislada y es mucho más fácil centrarse en que sea accesible para muchos tipos de usuarios.

CONCURRENCIA



La concurrencia es un problema que a menudo se presenta dentro del desarrollo de una aplicación, el tener en cuenta que hay cosas que quieren producirse al mismo tiempo no hay mucho problema porque los ID de desarrollo aceptan por medio de configuraciones cierta cantidad de concurrencia en la información, el problema está cuando se quieren realizar procesos muy pesados y que demoran mucho tiempo y el software debe continuar su flujo, o se tiene que hacer 2 o 3 operaciones concurrentes muy pesadas y el software tiene que seguir avanzando, es que donde se presenta el problema. El patrón arquitectónico donde se puede implementar este requisito no funcional es el patrón maestro esclavo, gracias a un conjunto de componentes aislados, es posible completar muchos trabajos pesados de manera concurrente y beneficiando directamente al performance de la aplicación, ya que el componente maestro, distribuye a un conjunto de componentes hijos a que realicen el trabajo que a él le tocaría solo.

Conclusión

En los diferentes tipos de requerimientos no funcionales podemos ver que es muy difícil lograr acoplarlos a un solo patrón dada su naturaleza, esto nos lleva a ser más analíticos desde el principio, con lo que se vuelve parte importante a la hora de diseñar, teniendo muy en cuenta y con claridad que es lo que se quiere lograr para no incurrir en contrariedades en los patrones.