Contents

Refactorizacion

Temas

- Que es
- Objetivo
- Caracteristicas
- Malos olores
- Mejora metodos
- Mejora clases
- Mejora jerarquias

Explicacion

La refactorizacion es la tarea que se encarga de cambiar la **estructura del codigo** con el fin de simplificarlo y mejorar su compresion, de manera tal que el comportamiento observacional del codigo no cambie.

Su objetivo principal es cambiar el diseño plasmado en el codigo. Es decir, disminuir acoplamiento, incrementar cohesion y mejorar la aplicacion del principio abierto-cerrado.

Una de las cosas que se deben evitar a toda costa al refactorizar, es mezclar refactorizacion de codigo con la implementacion de funcionalidad nueva. Esto se debe a que uno de los riesgos que conlleva refactorizar es el romper funcionalidad existente. Sin embargo, podemos mitigar este riesgo utilizando pasos lo mas pequeños posibles al refactorizar y teniendo scripts de tests para la funcionalidad ya implementada.

Para saber si hace falta refactorizar, se pueden observar ciertos indicios llamados "malos olores". Porque si hay un mal olor, lo mas probable es que algo esté podrido. Unos de los malos olores mas comunes son: - Clases largas - Metodos largos - Codigo repetido - Etc.

Las refactorizaciones mas comunes, pueden clasificarse en 3 tipos: - Mejoras de metodos: - Extraer metodo: Cuando un metodo realiza demasiadas acciones, se busca dividirlo en varios metodos de manera tal que la signatura del metodo indique que es lo que hace. - Añadir/eliminar parametro: Muchas veces, los metodos tienen parametros extra que no se usan o parametros faltantes que estaria bueno tener. Esto se realiza con el fin de **mejorar** las interfaces.

- Mejoras de clases:
  - Extraer clase: Cuando una clase abarca demasiados conceptos, viene bien separarla en varias clases para así mejorar la cohesion de sus

metodos.
  - Mover atributo: Cuando una clase utiliza mucho un atributo de otra clase, muchas veces es conveniente mover dicho atributo dentro de la clase que más lo está usando.
  - Mover metodo: Lo mismo sucede cuando una clase usa mucho un metodo de otra clase. Conviene moverlo a la clase que lo usa.
  - **Convertir dato en clase: Muchas veces se realizan muchas operaciones sobre conjuntos de datos, convirtiendose estos en una unidad logica. En estos casos lo mejor es crear una clase para estos datos, y añadirle metodos para las operaciones mas comunes que se realizan sobre ellos.**
- Mejora jerarquias:
  - Cambiar condicional por polimorfismo: Si se tienen condicionales muy extensos no se está aprovechando al máximo la capacidad de la programacion orientada a objetos y conviene crear una jerarquia de clases apropiada para reemplazar el condicional que se está usando.
  - Subir metodos/atributos: Si observamos que muchas clases hijas compartes los mismos metodos o atributos lo que debemos hacer es mover estos metodos o atributos a la clase madre.

Procesos codificacion

Codificacion Incremental

Pasos del proceso de codificacion incremental:

1. Se escribe codigo para una parte de la especificacion **(mencionar funcionalidad)**
2. Se escriben tests para el codigo **(solo codigo nuevo)**
3. Se ejecutan los tests **(Errors?)**
   a. Si pasan los tests: avanza al siguiente paso.
   b. Si no pasan: Se arregla el codigo y se vuelve al paso 3
4. Se ve si hace falta implementar algo mas de la especificacion:
   a. Si hace falta: vuelve paso 1
   b. Si no hace falta: se termina

De a pares

En la programacion de a pares dos programadores trabajan en conjunto siguiendo determinados roles. Uno de los programadores se centra en escribir el codigo, mientras que el otro se centra en analizar lo que se está escribiendo. Estos roles se van intercambiando periodicamente. **(PARTE DE AMBOS)**

Una de las ventajas de este tipo de proceso es que mejora:

- Algoritmos
- Estructuras de datos

- Contemplacion de casos
- Calidad general del codigo.
- **(REVISION DE CODIGO)**
- **(MEJOR DISEÑO)**

Sin embargo, aun no está bien en claro su efectividad, ya que puede que reduzca la productividad total de los programadores.

Procesos codificacion 2

Codificacion Incremental

La codificacion incremental es un proceso de codificacion que consta de los siguientes pasos: Partiendo de la especificacion:

1. Se escribe codigo para una parte de la funcionalidad
2. Se escriben scripts de tests para testear este nuevo codigo
3. Se corren los scripts de tests
4. Hubo errores?
    a. Si: Se arreglan los errores y se vuelve al paso 3
    b. No: Se continua al siguiente paso.
5. Se terminó con toda la especificacion?
    a. Si: Se termina el proceso
    b. No: se vuelve al paso 1

De a pares

La codificacion de a pares es un proceso de codificacion que consiste en que dos programadores van escribiendo el codigo de manera conjunta, siguiendo ciertos roles especificos. Uno de los programadores va escribiendo el codigo, mientras que el otro va analizando el codigo que se está escribiendo. Y en conjunto van pensado:

- Logica
- Algoritmos
- Estructuras de datos a usar.
- **(ESTRATEGIAS)**

Estos roles cambian periodicamente.

Este proceso cuenta con las siguientes ventajas: Se produce mejor **(diseño)**:

- logica
- algoritmos
- uso estructuras de datos
- contemplacion de posiblese casos Otra de las ventajas es que el codigo está bajo constante revision.

En cuestion de efectividad, esta no ha sido demostrada, ya que es posible que este proceso cause perdida de productividad, ya que dos programadores trabajan en una única tarea.

Modelo cascada

El modelo de cascada divide al proyecto en las siguientes fases:

1. Analisis de requerimientos y especificacion
2. Diseño de alto nivel
3. Diseño detallado
4. Codificacion
5. Testing
6. Instalacion

Cada fase comienza cuando finaliza la anterior, tiene una salida definida y se encarga de una incumbencia particular **(distinta)**.

Ventajas:

- Simple
- Facil de implementar
- Logico e intuitivo

Desventajas:

- Todo o nada: o se llega al resultado esperado o no se llega a nada
- Congelacion temprana de requerimientos
- No permite:
  - Cambios
  - Feedback del usuario
- La tecnologia o hardware elegido en un inicio puede terminar siendo viejo

Aplicacion:

- Proyectos donde:
  - Los programadores estan familiarizados con la metodologia
  - Los requerimientos son bien comprendidos
- **(problemas conocidos)**
- Proyectos corta duracion
- Se realiza la automatizacion procesos manuales existentes

Modelo prototipado

El proposito principal del modelo de prototipado es lograr una mejor compresion de los requerimientos. Para lograr esto se contruye un prototipo del producto final que luego debe descartarse.

(permite cliente mejor feedback) La etapa de analsis se reemplaza con una "mini-cascada" cuyas fases no son ni muy exhaustivas ni muy rigurosas **(formales, minuciosas)**

En la etapa de desarrollo el foco principal son las características que no se comprenden muy bien. Por lo cual no se desarrollan las caracteristicas que si son bien entendidas. Se muestra al usuario el prototipo, este juega con el, lo cual le permite dar un buen feedback sobre que es lo que quiere. Una vez el usuario brinda su feedback, este se implementa y se repite el proceso. Esto continua hasta que el costo supere al beneficio de tener un prototipo. Una vez termina el prototipado, se toma lo aprendido durante esta etapa **(modifican req iniciales)** y se plasma en la especificacion final.

El costo justamente es un factor muy importante, porque en el prototipado lo que se busca es reducirlo al máximo, ya sea no implementando características a no ser que sea absolutamente necesario, reduciendo la cantidad de tests, o la aplicacion de estandares. **(calidad no importa)** Todo lo que haga falta para aumentar la velocidad de desarrolló y así reducir el costo. Ya que el costo del prototipado debe ser solo un pequeño porcentage del costo total.

Ventajas:

- Se comprenden mucho mejor los requerimientos
- **(recoleccion req)**
- Los desarrolladores ganan experiencia util que servirá para el desarrolló del proyecto real
- **(reduce riesgo)**
- Se obtiene muy buen feedback del usuario
- **(sistemas finales mejores, mas estables)**
- Produce requerimientos mucho mas estables

Desventajas:

- **(comienzo pesado)**
- Es complicado cambiar requerimientos una vez definidos **(MAL)**
- Puede aumentar costo y tiempo total
- **(no permite cambios tardios)**

Aplicacion:

- **(usuarios novatos)**
- Proyectos donde los requerimientos son inciertos
- Proyectos donde las interfaces de usuario son muy importantes

Modelo iterativo

Caracteristicas

- Aborda el problema “todo o nada” de cascada produciendo software incrementalmente.
- **(combina beneficios prototipado cascada)**
- En cada iteracion se implementa un subconjunto de la especificacion.
- **(cada incremento es completo)**
- Provee un buen marco para realizar testing, ya que este se realiza de manera incremental.
- Se puede obtener un muy buen feedback del usuario, ya que es posible probar
- las implementaciones parciales
- **(feedback cada iteracion iteracion siguiente)**

LCP

- Es la lista de control del proyecto.
- Lleva registro de las tareas necesarias para el proyecto.
- **(registro en orden)**
- **(guia pasos iteracion)**
- **(cada entrada)**

Proceso

- **(Crear LCP)**
- En cada etapa **(paso)** se eligen tareas del LCP y se realiza: planificacion, analisis
- **(eliminar siguiente tarea lista)**
- **(implementacion)**
- **(del sistema parcial)**
- **(proceso se repite vaciar lista)**
- para llevar a cabo esas tareas

Fortalezas

- **(entregas)**
- **(reduce riesgo)**
- **(acepta cambios)**
- **(prioriza requisitos)**
- Buen feedback
- Bueno para proyectos largos

Debilidades

- Sobrecarga de planificacion de cada fase
- Costo y tiempo total pueden ser mas elevados
- **(arquitectura, diseño pueden sufrir)**

Aplicacion Proyectos que deben desarrollarse de manera rapida

- **(timpo es esencial)**
- **(riesgo proyectos largos)**
- **(requerimientos desconocidos)**

- Proyectos que tengan que adaptarse a cambios

Modelo iterativo 2

El modelo iterativo es un modelo de proceso de desarrollo donde el software se va implementando de manera incremental. Se busca abordar el problema "todo o nada" de cascada y se mezclan lo beneficios de cascada y prototipado. Se va elaborando el software en incrementos que son completos por si mismos. Se van implementando subconjuntos de la especificacion total. Por lo cual, se facilita obtener feedback del usuario y de el rendimiento de los procesos, el cual puede ser usado en las futuras iteraciones. Tambien provee un modelo muy interesante para el testing, ya que como se va desarrollando el software en incrementos, es mucho mas sencillo hacer tests que si se hiciesen para el software completo.

Un concepto muy importante dentro de el modelo de codificacion iterativo es la lista de control del proyecto (LCP). La cual guarda las tareas necesarias (en orden) para que el proyecto sea completado. Cada entrada de la LCP debe ser lo suficientemente simple para ser comprendida en su totalidad.

Entonces los pasos para llevar a cabo el proceso son los siguientes: Primero se crea la LCP. Y por cada paso se van eliminando tareas de la LCP a su vez que se va haciendo diseño, implementacion y analisis del sistema parcial. El proceso termina una vez ya no hay mas tareas en el LCP.

Fortalezas:

- Entregas rapidas e incrementales **(regulares)**
- Se adapta muy bien a cambios
- Se puede obtener muy buen feedback de los usuarios
- Menor riesgo que otros modelos
- prioriza requisitos

Debilidades:

- Sobrecarga debido al planeamiento de cada fase **(iteracion)**
- Puede ser mas costoso que otros metodos
- El diseño y la arquitectura pueden sufrir debido a los cambios que se vayan realizando en cada iteracion.

Aplicacion: - Empresas que busquen reducir el time to market. - Empresas que no puedan permitirse el riesgo de un proyecto demasiado largo. - Cuando los requisitos son inciertos.

Porque separar en fases?

- Utiliza el principio divide y venceras. Es mucho más fácil atacar una serie de problemas chicos que uno problema grande.
- Cada fase tiene una incumbencia distinta

- **(cada fase distintas partes problema)**
- Se puede validar el resultado de cada fase para encontrar errores lo mas temprano posible.

Porque separar en fases? 2

- Utiliza el principio divide y venceras
- Cada fase ataca distintas partes del problema
- Permite validar continuamente el proyecto

Enfoque ETVX

El enfoque ETVX es el enfoque que se utiliza en las fases del proceso de desarrollo y consta de las siguientes partes: - Criterio de entrada (E): Cuando se debe iniciar la fase **(condiciones)** - Tarea (T): la tarea a realizar en la fase - Validacion (V): Incluye las inspecciones, revisiones y validaciones que se realizan **(controles)** a la salida de la fase, es decir, que se realizan al producto de trabajo. - Criterio de salida (X): Determina cuando la fase se puede considerar como terminada **(produce informacion admin proceso)**

Enfoque ETVX 2

- Verificacion (V): incluye las inspecciones, controles, revisiones, verificaciones que se hacen sobre la salida de la fase, es decir, sobre el producto de trabajo Cada fase produce informacion relevante para la administracion del proceso

Fases proceso administracion proyecto

Planeacion:

- Se realiza antes de comenzar el proyecto
- Tareas principales:
  - Estimacion de tiempo y costo
  - Seleccion de personal
  - Planeacion de seguimiento
  - Planeacion del control del calidad

Seguimiento y control:

- Acompaña al proceso de desarrollo
- Tareas principales
  - Realizar seguimiento de parametros relacionados a tiempo, costo, riesgos y los factores que los afectan
  - tomar accion correctiva de ser necesario.

- Las metricas aportan los datos necesarios para el seguimiento

Analisis de terminacion:

- Se realiza una vez finaliza el desarrollo
- Su proposito es:
  - Analizar posibles mejoras del proceso **(desempeño proceso)**
  - Identificar lecciones aprendidas

En procesos iterativos esta estapa se realiza al final de cada iteracion Se utiliza lo aprendido para mejorar la iteracion siguiente

Fases proceso administracion proyecto 2

Analisis de terminacion: - Proposito - Analizar desempeño del proceso - Identificar posibles lecciones a aprender

Oraculo de tests

Muchas veces para verificar la existencia de un desperfecto necesitamos conocer el comportamiento esperado de un programa. En esos casos entra en el juego el oraculo de test. El oraculo de tests es quien define el comportamiento esperado para un programa. Muchas veces el oraculo es humano, por lo tanto es propenso a cometer errores. Un oraculo humano utiliza la especificacion para determinar el comportamiento esperado. Sin embargo, esto no es infalible ya que la especificacion tambien puede cometer errores

Criterios de seleccion de test

Idealmente, nos interesa que la ejecucion completa **(satisfactoria)** del conjunto de casos de tests implique una ausencia de defectos. Sin embargo, como los tests son caros, nos interesa que este conjunto sea lo mas reducido posible. Aquí es donde entran en juego los criterios de seleccion de tests, los cuales son una serie de condiciones que el conjunto de tests debe cumplir con respecto al programa y/o a la especificacion

Intento 2

Idealmente, nos intereza que la ejecucion satisfactoria del conjunto de tests implique una ausencia de defectos

Cocomo

El sistema COCOMO tiene los siguientes pasos:

A) Estimador inicial

Se calcula lo siguiente: Ei = a * tamañoˆb

Donde a y b toman valores específicos dependiendo de si el proyecto es organico, semi-rigido u rigido.

Un proyecto es organico cuando es poco estructurado y se trabajo con equipos pequeños.

Un proyecto es rigido cuando es ambicioso e innovador, tiene altas restricciones impuestas por el entorno y tiene altos requerimientos de confiabilidad e interfaces.

B) Calcular 15 factores de multiplicacion para el esfuerzo

Se calculan 15 factores de multiplicacion fk. Los cuales pueden pertenecer a distintos tipos y tomar uno de 6 valores de importancia que van desde muy bajo hasta muy alto. Segun el valor de importancia que se le haya asignado se le asigna un factor de ajuste.

C) Ajustar estimacion de esfuerzo

Teniendo el valor de Ei del punto (A) y los fk del punto (B), calculamos el esfuerzo de la siguiente forma:

esfuerzo = $E_i * \Pi_{k=1}^{15} f_k$

D) Determinar el porcentage de esfuerzo de cada una de las fases

En base al tamaño del proyecto, se le asigna un porcentage de esfuerzo a cada una de las fases, las cuales pueden ser:

- Diseño del producto
- Diseño detallado
- Codificacion y unit test
- Integracion y test

(tabla distinta segun tipo sistema) Se usa una tabla distinta segun el tipo de sistema

Particionado clase equivalencia

Se divide el espacio de entrada en clases de equivalencia. Cada condicion especificada como entrada es una clase de equivalencia

La idea general es que si un caso de test es satisfactorio para un elemento de una clase de equivalencia muy probablemente lo sea para el resto de elementos de esa clase.

Para añadir mas robustez algoq que se puede hacer es crear clases de equivalencia para valores invalidos y para datos de salida.

Ahora se pueden usar dos criterios para realizar los tests: 1. Se hacen tests para cada uno de las clases de equivalencia **(tantas como sea posible)** 2. Se hace a lo sumo un test para cada clase de equivalencia valida. **(por cada entrada)**

Analisis de valores limites

La gran mayoria de errores suelen encontrarse justo fuera o justo dentro de los limites.

Debido a esto, por cada clase de equivalencia, tomamos valores que esten justo dentro o justo fuera de la clase Y hacemos algo similar para los datos de salida, buscamos que se produzcan valores en los limites de la clase de equivalencia.

Que hacer si tenemos multiples entradas? Hay dos formas de afrontar esto:

1. Testear todas las combinaciones posibles. Lo cual implica escribir muchas tests, lo cual es caro.
2. Testear valores limites para una entrada a la vez y que el resto de las entradas tomen valores nomales

Coberturas

De sentencia

Toda sentencia debe ejecutarse al menos una vez durante el testing

De ramificaciones

Cada decision debe ejercitarse como verdadera y falsa durante el testing.

Implica cobertura de sentencia.