UNIVERSIDAD DE BURGOS

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Simulación de un proceso de refactorizacioón.

Alumnos	María Portugal Tomé Marcos Romano Ibáñez Guillermo Saldaña Suárez Jesús González Alonso
Tutor	Carlos López Nozal DEPARTAMENTO DE INGENIERÍA CIVIL Área de Lenguajes y Sistemas Informáticos

Burgos, 20 de mayo de 2021

ÍNDICE

1 OBJETIVOS

Aplicar un proceso de refactorización sobre un pequeño proyecto existente

- Conocer como corregir defectos de código usando refactorizaciones
- Relacionar una lista de tareas de desarrollo y de mantenimiento software con el proceso de refactorización.

2 CONTEXTO

Estas manteniendo un sistema software que representa una simulación de una red de área local (LAN). El equipo de desarrollo ha sido muy rápido en adaptar los requisitos iniciales para el sistema entregando una versión 1.4 que contiene la funcionalidad para el primer hito. El cliente solicita añadir una nueva funcionalidad y el equipo de desarrollo se percata que el diseño no está preparado.

Saben que eres un experto en refactorización, por eso te prestan su código para que lo refactorices apropiadamente. No esperan un diseño perfecto, esperan un diseño que permita añadir la nueva funcionalidad fácilmente. Además, tienen disponibles pruebas del sistema desarrollado.

3 TAREAS A REALIZAR.

3.1 Ojear la documentación (OORP – p. 44)

¿Cuál es tu primera impresión sobre el sistema? ¿Dónde centrarías tus esfuerzos de refactorización? Discutir con los miembros del equipo.

- Del fichero *"LANSimulationDocu_es.pdf"* podemos obtener una visión muy general de sistema, así como una relación entre los diferentes paquetes y clases.
- Del fichero *"toDoList_es"* podemos obtener un historial de las implementaciones realizadas y las que quedan por implementar.
- Y de la documentación de Java/doc podemos obtener una visión más concreta de lo que se ha implementado.

En un principio parece que la estructura de paquetes y relaciones es aceptable con la funcionalidad del sistema.

Teniendo en cuenta los ficheros anteriores y la documentación, parece que el paquete que puede dar más problemas sea el de *"lanSimulacion"*, ya que es el encargado de realizar toda la gestión de las diferentes opciones que provee el sistema. Además, este paquete y sus clases (y los métodos de estas) carecen de la documentación necesaria para saber qué es lo que se está haciendo en cada uno de estos componentes.

3.2 Lee todo el código en 5 minutos (OORP – p. 38)

¿Cuál es la segunda impresión sobre el sistema? ¿Estás de acuerdo con la impresión inicial? Con este nuevo conocimiento sobre el código. ¿dónde centrarías tus esfuerzos de refactorización? Discutir con los miembros del equipo.

Tras haber mirado el código un poco por encima, nuestras sospechas iniciales se confirman.

Por ejemplo:

- El fichero *"LANSimulation.java"* tiene un método *"simulate()"* que es interminable, demasiado largo, por lo que es muy probable que durante la refactorización se extraigan unos cuantos métodos de este.
- Cabe destacar en cada una de las clases/métodos la documentación deja mucho que desear, ya sea por su inexistencia o por la carencia de estructura al generarla.
- Además, el código no tiene ningún tipo de formato, lo cual hace más difícil su comprensión.

3.3 Hacer una instalación de prueba (OORP – p. 58)

¿Crees que el código base esta ya refactorizado? ¿Qué puedes decir de la calidad de los tests: puedes empezar a refactorizar de manera segura? Discutir con los miembros del equipo.

El código base no está refactorizado, con una vista previa se puede observar que existen elementos/compontes a los que hay que aplicar una gran variedad de refactorizaciones.

Por otro lado, tras la ejecución de los test no podemos empezar a refactorizar de forma segura, ya que aparte de que estos no cubren todo el código, además hay un test que no es superado por el código implementado.

Así que antes de empezar a refactorizar hay que poner los test a punto, para que a la hora de empezar a refactorizar el comportamiento del sistema no cambie.

3.4 Habla con los de mantenimiento (OORP – p. 31)

Desarrolla un plan de proyecto listando a) los riesgos, b) las oportunidades de refactorización (detección de defectos), c) las actividades (plan de refactorizaciones).

3.4.1 Riesgos

- Como la cobertura de los test no es del 100% sobre todos los paquetes, clases y métodos, nos podemos arriesgar a cambiar la funcionalidad del sistema sin ser conscientes de ello.

3.4.2 Las oportunidades de refactorización (detección de defectos)

- Como se ha mencionado anteriormente, en este sistema se pueden detectar una gran cantidad de defectos, como *Long Method*, *Large Class*, *Duplicated Code*, *Feature Envy* entre otros y por lo tanto una gran variedad de oportunidades para la refactorización del código y de este modo conseguir que el sistema sea más compresible y adaptable sin modificar su comportamiento externo.

3.4.3 Las actividades (plan de refactorizaciones)

- Tras haber detectado los principales defectos se proseguirá con la búsqueda de una solución a estos sin que se modifique el comportamiento externo del sistema.
  En el apartado cuatro se puede ver con detalle las refactorizaciones realizadas.

3.5 Extract Method (OORP – p. 255)

Una de las cosas que podrías haber visto es que hay una considerable cantidad de código duplicado que representa una importante lógica de dominio dentro de la clase Network. Decides primero deshacerte del código duplicado.

- El código de “accounting” ocurre dos veces dentro de "printDocument". Elimina las copias aplicando EXTRACT METHOD.
- El código de “logging” ocurre tres veces, dos dentro de "requestWorkstationPrintsDocument" y una dentro de " requestBroadcast". Elimina las copias aplicando EXTRACT METHOD. Nota que las tres copias no son exactamente la misma por eso tendrás que adaptar el código un poco antes de hacer la refactorización.
- ¿Hay más código duplicado representando la lógica de dominio que debería ser refactorizado? ¿Puedes refactorizarlo usando EXTRACT METHOD?
  - Existen tres métodos diferentes en la clase Network que utilizan una estructura de *switch* similar entre ellos, pero cada uno de ellos alguna diferencia, estos métodos son:
    - *printOn.*
    - *printXMLOn.*
    - *printHTMLOn.*
  - Pero si la refactorización se realizaría con *Extract Method* el método resultante debería de poder diferenciar entre las tres posibles representaciones y actuar en consecuencia.
  - Aunque si la refactorización se hiciese únicamente sobre los métodos *printOn* y *printHTMLOn* sí que se mejoría, pero también existe la desventaja de que en un futuro solo se quiere modificar la representación de uno de ellos volveríamos a tener que hacer una

diferencia entre ambas representaciones dentro de un mismo método, lo cual no es que mejore considerablemente el estado actual.

¿Estás seguro que estas refactorizaciones no rompen el código? ¿Crees que estas refactorizaciones merecen la pena? ¿La herramienta de refactorización hace un buen trabajo? Discutir con los miembros del equipo.

Las refactorizaciones realizadas a lo largo de este apartado no rompen el código, ya que tras cada una de las modificaciones que se han realizado se han ejecutado los test y según estos el comportamiento externo del sistema no ha variado.

Como se ha indicado anteriormente hay algunas refactorizaciones que sí que merecen la pena realizarlas ya que eliminan una gran cantidad de código duplicado, pero por otra parte también existen otras que no son tan buenas ya que puede que un futuro cuando se quiera ampliar el sistema esta refactorización complique el proceso.

La herramienta de refactorización sí que hace un buen trabajo siempre y cuando se utilicen de la forma adecuada, teniendo en cuenta la futura extensión del sistema.

3.6 Mover el comportamiento cerca de los datos (OORP – p. 190)

Tomando los métodos extraídos, nota que ninguno de ellos se refiere a atributos definidos en la clase Network. Por otro lado, estos métodos acceden a campos públicos de las clases "Node" y "Packet".

- El método “logging” que acabas de extraer no pertenece a Network porque muchos de los datos a los que accede pertenecen a otra clase. Aplicar MOVE METHOD para definir el comportamiento cerrado de los datos con los que opera un método.
- De la misma manera, el método "printDocument" accede a atributos de otras clases, pero no accede a sus propios atributos.
- ¿Hay más métodos que pueden ser movidos junto con los datos con los que operan? Si existen aplica MOVE METHOD hasta que estés satisfecho con los resultados.
    - En los métodos print de Network las instrucciones que guardan cadenas en el buffer se han movido a Node, pues no solo hay duplicación de características, sino que además esas porciones de los métodos muestran envidia de características de Node.

3.7 Eliminar código de navegación (OORP – p. 199)

Existe todavía una porción de código duplicado, si se sigue la pista del puntero "nextNode_" hasta que se recorre completamente la red; la lógica esta duplicada en "requestWorkstationPrintsDocument" y "requestBroadcast" (y con menos grado en "printOn", "printHTMLOn", "printXMLOn"). Esta lógica duplicada es bastante vulnerable, porque accede a atributos definidos en otras clases y de hecho representa una clase especial de código de navegación.

- Aplicar EXTRACT METHOD sobre la expresión booleana definida al final del bucle creando un método predicado "atDestination".
- Rescribe los bucles dirigidos por "currentNode = currentNode.nextNode_" dentro de las llamadas recursivas de un método "send".

- ¿Estás seguro que estas refactorizaciones no rompen el código? ¿Crees que estas refactorizaciones merecen la pena? ¿La herramienta de refactorización hace un buen trabajo? Discutir con los miembros del equipo.
    - No rompen el código. Se puede comprobar lanzando los test unitarios. Adicionalmente, los cambios que se están realizando son refactorizaciones, cuya característica es que no cambian la ejecución del programa.
    - Estas refactorizaciones principalmente reducen la duplicación del código. Estos cambios merecen la pena en el momento en el que haya que hacer un cambio en ese método y solo haya que hacer un cambio sobre el método extraído y no sobre todas las líneas en las que se ha aplicado ese método extraído, ahorrando tiempo de mantenimiento y esfuerzos de búsqueda y actualización de esas instrucciones (en caso de que el método no hubiese sido extraído).
    - La herramienta de refactorización realiza satisfactoriamente la refactorización. Sin embargo, limita mucho la customización del método impidiendo añadir o eliminar parámetros o modificar el valor que se retorna. Un ejemplo de cuando esto puede ser molesto es en caso de que una clase tuviese un atributo de la clase X y al hacer una comparación (por ejemplo) del tipo "X variable == X atributo", al extraer el método no puede hacer que la comparación sea entre dos variables de tipo X (pasando dos parámetros de tipo X) sino que solo se puede pasar un parámetro que automáticamente se comparará con el atributo X de la clase.

3.8 Transformar códigos de tipo (OORP – p. 217)

Otra porción de código duplicado se puede encontrar en "printOn", "printHTMLOn", "printXMLOn". Esta vez se trata de un duplicado condicional, el código es probable que cambie dada la funcionalidad extra de introducción de un nodo GATEWAY. Por esto merece la pena introducir una nueva subclase aquí:

- Deberías haber notado que al mover el comportamiento cerca de los datos, que las sentencias switch dentro de "printOn", "printHTMLOn", "printXMLOn" deberían haber sido extraídas y movidas sobre la clase Node. Si no lo hiciste hazlo ahora, nombrando los nuevos métodos "printOn", "printHTMLOn", "printXMLOn".
- Crea subclases vacías para los diferentes tipos de nodo ("WorkStation", "Printer").
- Ajusta la invocación de los clientes a los constructores para que creen instancias de la clase apropiada.
- Mueve el código desde cada cuerpo de la sentencia condicional a la (sub)clase apropiada eliminando eventualmente el condicional.
- Verifica todos los accesos al atributo "type_" de Node. Mientras encuentres alguna referencia elimínala.
- Elimina el atributo "type_".
- ¿Estás seguro que estas refactorizaciones no rompen el código? ¿Crees que estas refactorizaciones merecen la pena? ¿La herramienta de refactorización hace un buen trabajo? Discutir con los miembros del equipo.
    - Estas refactorizaciones (como todas) no tienen el objetivo de cambiar el funcionamiento externo. Sin embargo, este cambio es ligeramente complejo y es fácil romper el código haciendo el cambio.

- Comprimir en una sola clase (Node) las tareas de clase con los mismos métodos, pero con distintas implementaciones de este hace la clase mucho más compleja de lo que en verdad debería ser (además de romper el principio SOLID open/close porque no hay otra forma de afrontar ese método) y con atributos innecesarios (type_) que complican el constructor y la implementación de los métodos. Esto es muy probable que alargue el tiempo de trabajo cuando toque hacer cambios sobre la clase Node además de obligar a efectuar cambios sobre esa clase que no corresponden a esa clase (corresponden a Printer y Workstation).
- En este caso la herramienta de refactorización no ha sido inútil y ha sido necesario realizar la refactorización a mano. Es comprensible que la herramienta no haya sido capaz de manejar esta refactorización, ya que la herramienta de refactorización como mucho puede realizar los pasos necesarios para crear una clase hija, pero no todas las operaciones tangentes a extraer un hijo pero que la herramienta no puede deducir.

4 HISTORIAL DE REFACTORIZACIONES POR CLASES

4.1 Clase LANTests

4.1.1 Método YOUMAYWANTTOtestCompareFiles.

- Muestra el defecto código muerto. Se eliminará ese método sin usar.

4.1.2 Método compareFiles.

- Los nombres de las variables no son claros. Se van a sustituir los nombres de las variables por nombres que representen lo que contienen.
- Comentarios redundantes intentando explicar que sucede. Estos comentarios son necesarios porque no se sabe que representa cada variable. Después de renombrar las variables se borrarán los comentarios redundantes.

4.2 Clase LANSimulation

4.2.1 Método Simulate

- Muestra el defecto LongMethod. Método muy largo que no ha sido dividido en submétodos y con mucha duplicación de código. Recomendable dividir en submétodos Simulate y plantear si se puede escribir un solo método que con una serie de parámetros agrupe varias cadenas de instrucciones en un solo método. Se van a extraer submétodos que aclaren que está haciendo el método.
- Muestra el defecto envidia de características. El método se dedica casi exclusivamente a llamar métodos se System.Out. Sin embargo, no es que haya una forma más limpia de realizar estas operaciones. No se realizará ninguna refactorización con el objetivo de solucionar este defecto.
- Muestra el defecto Magic number pero con cadenas y no números. Hay una serie de cadenas que se utilizan repetidas veces en el método, pero no se guardan en una variable. Se van a meter en una variable.

4.2.2 Método Main

- Muestra el defecto de comentarios innecesarios en las líneas 124, 126 y 136. Separando las operaciones a las que hace referencia estos comentarios en submétodos con un nombre lo suficiente mente esclarecedor y eliminando los comentarios debería ser suficiente.

4.3 Clase Network

- La declaración de las variables no sigue el convenio establecido, ya que las únicas variables que pueden tener un nombre terminado con "_" son las variables *final*. Por ello se va a realizar un renombrado de dichas variables.

4.3.1 Constructor

- Durante la construcción del objeto, se comprueba si el objeto ya ha sido inicializado, lo cual es redundante, porque tras construirlo compruebas que se ha construido. Por ello, se eliminará esta comprobación del constructor.
- Además, tras la construcción del objeto también se comprueba si no es una red consistente. Una red no es *consistent* cuando las *workstations* están vacías, que es el caso inicial, cuando creamos el objeto. Por lo tanto, como ya se define que inicialmente las *workstations* están vacías no es necesario comprobar posteriormente si se trata de una red no *consistent*, por esto se eliminará esta comprobación del constructor de la clase.

4.3.2 Método DefaultExample

- El nombre de este método no sigue el convenio establecido, ya que los nombres de los métodos no deberían empezar con una letra mayúscula, por esto, se va a realizar un renombrado de método para ajustarnos al mencionado convenio.
- También nos encontramos con un conjunto de variables que son necesarias para definir un objeto *Network*, y como estas variables son inmutables a lo largo de este método se las puede establecer como *final.*

4.3.3 Método isInitialized

- Para realizar la comparación entre dos objetos se debe utilizar *equals()* no *"=="*, así como la utilización de paréntesis innecesarios.

4.3.4 Método hasWorkstation

- El nombre de la variable pasada por parámetro es poco descriptivo, renombrar. Además, como esta variable no se va a modificar a lo largo del método, se puede establecer como *final*.
    - *"ws"* → *"workstation"*
- La variable definida *"n"* también es poco descriptiva, renombrar.
    - *"n"* → *"node"*
- Eliminación de la parte *else*, no es necesaria para el correcto funcionamiento.
- Eliminación de comentario innecesario (línea 112).

4.3.5 Método printDocument

- Existe código duplicado, exactamente el código de "accounting", por lo tanto, se va a realizar un *Extract Method* y se sustituirá el mencionado código duplicado por una llamada a este nuevo método *"accountingDocument"*, además se añadirá una distinción ya que el código varía ligeramente.

    - Líneas con la nueva llamada al método: 347 y 354.
- Adicionalmente este método utiliza muchos atributos de clases ajenas y pocas de la clase Network. Es por ello que se va a aplicar la refactorización *Move Method* para mover el método de Network a Package.

4.3.6 Método logging

- El método logging de Network utiliza muchos atributos de otras clases y ninguno de esta. Es por ello que se va a aplicar *Move Method* para mover el método logging de Network a Node.

4.3.7 Método requestWorkstationsPrintsDocument y requestBroadcast

- Existe código duplicado dentro de estos métodos, por ello se va a extraer dicho código en un método llamado "logging", teniendo en cuenta las pequeñas diferencias entre ambos métodos se adaptará el código con una sentencia condicional.
- Otra medida adicional que se ha tomado ha sido extraer tambien el método "atDestination" para reducir la duplicación de código en los bucles while.

4.3.8 Métodos printOn, printHTMLOn y printXMLOn

- La comprobación de los whiles de estos métodos es la misma. Se va a extraer un método general de estas comprobaciones del while para reducir la duplicación del código.
- Las instrucciones dentro del switch se han movido a Node pues muestran envidia de características de Node y duplicación de código entre los métodos, que tienen las mismas instrucciones.

4.3.9 Otros

- En la clase Network hay una serie de "catch" vacíos con el comentario "//just ignore". Estos comentarios son innecesarios y se han eliminado.
    - En el método requestBroadcast.
    - En el método requestWorkstationPrintsDocument.
    - En el método logging (previo al *Move Method* a Node).
- Hay una instrucción que se repite numerosas veces en distintos métodos. Se ha extraído esa instrucción al método Network.send para reducir la duplicación de código.

4.4 Clase Packet

- Renombrado los nombres de las variables ya que estos no siguen el convenio establecido.

4.4.1 Método printDocument

- Eliminación de puntos y comas (;) innecesarios.
- Muestra defecto de duplicación de código. Para solucionarlo, hemos realizado un *Extract Method* de los *if(s)* dentro de la primera condición del try, creando el método *generateText*. Este método ha sido modificado manualmente para mantener el comportamiento externo de la aplicación.
- Renombrado del parámetro *author* a *typeMessage* en el método *generateText*.

4.5 Clase Node

- Se han extraído las clases hijas de Node: Workstation y Printer. Como ya no es necesaria la variable type_ que indique de que tipo es cada nodo se ha eliminado.

5 PREGUNTAS DE REFLEXIÓN

¿Se puede automatizar completamente el proceso de refactorización a través de herramientas?

No se puede automatizar completamente el proceso de refactorización ya que nos hemos encontrado en el código defectos que no pueden solucionarse con las herramientas de eclipse. Como, por ejemplo, puntos y coma (;) que no estaban asociados al final de una sentencia.

Además, en clase se ha mencionado que existen procesos de refactorización como por ejemplo Encapsulate Field que deben realizarse manualmente.

Y con estos ejemplos queda demostrado que no te puede automatizar totalmente el proceso de refactorización.

Cabe mencionar que hay refactorizaciones que dependen por completo de la subjetividad, como por ejemplo la refactorización rename. Un programa no puede tener una idea clara de por qué nombre cambiar el actual de un método, clase o variable.

¿Qué relación encuentras entre el proceso de refactorización y la utilización de sistemas de control de tareas y versiones?

El proceso de refactorización va íntimamente ligado al de control de tareas y versiones, pues las refactorizaciones están pensadas para realizarse cuando se añade una nueva funcionalidad o se buscan fallos. Es recomendable refactorizar antes de realizar estas tareas para facilitar el trabajo que viene a continuación y que puede surgir en un futuro no inmediato.

Sin embargo, no se puede estar refactorizando indefinidamente, pues no se avanzaría en el proyecto. Así que hay que decidir cuál es el momento correcto para aplicar refactorizaciones sin detener en exceso el desarrollo del programa.

El control de tareas y versiones es muy útil para gestionar cuando es recomendable hacer una refactorización y con qué objetivo.

6 CONCLUSIÓN

Una vez realizada la práctica podemos concluir que una serie de pequeños cambios pueden facilitar mucho el mantenimiento, así como la evolución del sistema. El proceso de refactorización en muchos casos puede llegar a ser realmente laborioso.

Las pruebas del código nos proporcionan cierta tranquilidad y garantizan que no se produzca un cambio en la funcionalidad.

Tener por último en cuenta que las refactorizaciones son un alto en el camino y no un fin en sí mismo, pues no afectan al comportamiento externo y no tiene sentid hacerlas si luego no se va a modificar el código.

7 REFERENCIAS

1. **[En línea] https://github.com/GuillermoSaldana/reafactoring_lab_session.**