Diseño y pruebas unitarias

02/01/2024

3º GEI
Enginyeria del Programari
Universitat de Lleida
David Carreras
Dídac Cayuela
Luis Carrasquer

Índice

CRITERIOS EMPLEADOS

- **principios SOLID**
- **patrones GRASP**

MÉTODOS DE REFACTORING como solución a CODE SMELLS

- ### Dead Code

  En el proyecto que hemos desarrollado, hemos utilizado una variable `scanner` que se inicializa pero no se utiliza en ninguna otra parte del código. Aunque esto podría considerarse un "Dead Code", hemos decidido mantenerla por una razón específica.
  Este code smell estaría catalogado en "Dispensables", lo que indica que es algo inútil e innecesario cuya ausencia haría que el código fuera más limpio, más eficiente y más fácil de entender.

  ```
  private final Scanner scanner;
  ```

  La variable scanner está ahí para representar simbólicamente que el usuario está insertando cosas en la terminal. Aunque no se utiliza en ninguna operación, su presencia ayuda a entender cómo se espera que funcione el código. Es como un recordatorio de que se espera una interacción del usuario.

  Entendemos que mantener este tipo de código puede llevar a confusión y desorden en la base de código. Por eso, hemos añadido comentarios en el código para explicar la presencia de la variable `scanner`. De esta manera, cualquier otro desarrollador que trabaje en este código en el futuro entenderá su propósito.

- ### Duplicate Code

  El "code smell" de "Duplicate Code" se refiere a la presencia de fragmentos de código idénticos o muy similares en diferentes partes de un programa.

  El método de refactoring que aplicamos es "**Extract method**". Antes de aplicar la refactorización "Extract method", teníamos condicionales repetidos en cada una de las funciones, lo que resultaba en código duplicado y dificultaba la comprensión del flujo del programa. Ahora cada función

(checkManualStep, checkBiomStep, incManualStep, incBiomStep) contiene condiciones específicas (opt == 'd', opt == 'n', opt == 'p') que determinaban si la función debe ejecutarse en función de si ya se han ejecutado los pasos previos o por el contrario no, y en consecuencia debe lanzarse la ProceduralException.

Quizás podría haberse hecho un refactoring un tanto más exhaustivo de la siguiente forma, pero consideramos que la opción que tenemos en el proyecto es más cómoda que la siguiente de cara al programador:

```
private void checkStep(int stepNumber, char opt, int procedureStep,
String errorMessage) throws ProceduralException {

   if ((opt == 'd' || opt == 'n' || (opt == 'p' && procedureStep
!= stepNumber))) {

       throw      new      ProceduralException(errorMessage      +
procedureStep);

   }

}

private void incStep(char opt, int procedureStep) {

   if ((opt == 'd' || opt == 'n' || opt == 'p')) {

       procedureStep++;

   }

}
```

Optamos finalmente por separarlo en funciones distintas en función de si se tratan de los pasos manuales o biométricos, ya que pese a continuar teniendo un poco de duplicación, nos resultaba más cómodo a la hora de utilizar dichas funciones.

- ## Large Class

El "code smell" de "Large Class" sugiere que la clase VotingKiosk puede tener demasiadas responsabilidades y podría beneficiarse de una refactorización. Una solución común para abordar este problema es aplicar el principio "Extract Subclass". Este principio implica dividir la clase principal en subclases más específicas, cada una encargada de una parte particular de la funcionalidad.

En el caso de VotingKiosk, identificamos dos áreas de responsabilidad principales: la votación manual y la verificación biométrica. Al aplicar "Extract Subclass", crearíamos dos nuevas subclases, ManualVotingKiosk y BiometricVotingKiosk, para manejar estas responsabilidades de manera más específica.

```
public class ManualVotingKiosk extends VotingKiosk {

   // Lógica específica para la votación manual

}
```

```
public class BiometricVotingKiosk extends VotingKiosk {

   // Lógica específica para la verificación biométrica

}.
```

En estas subclases, podríamos mover y refactorizar la lógica relacionada con la votación manual y la verificación biométrica, respectivamente. Luego, la clase VotingKiosk principal podría ser una clase más general que coordina el flujo general todo el proceso a lo largo de la votación.

Esto haría que cada subclase tenga una responsabilidad específica y facilitaría la extensión y el mantenimiento del código. Además, podríamos evitar tener múltiples variables de control de progreso (manualProcedureStep y biomProcedureStep) y simplificar la estructura general de la clase.

- ## CLASE EXTRA “Passport”

La clase Passport ha sido diseñada para modelar la información vinculada a un pasaporte, abarcando datos biométricos y el NIF (Número de Identificación Fiscal) del titular del pasaporte. Su objetivo principal es ofrecer una representación estructurada y coherente de los detalles inherentes a un pasaporte. Esto simplifica el acceso y la manipulación de los datos relacionados con la identidad y biometría del titular del pasaporte, ya que se observó una gran correlación entre dichos datos y fué creada con la finalidad de facilitar el manejo de estos.

- ## Magic Numbers

En el código de la clase VotingKiosk, se observa el uso de "magic numbers" para representar los pasos en los procedimientos manuales y biométricos. Aunque es una práctica común reemplazar estos números mágicos con constantes simbólicas para mejorar la legibilidad y el mantenimiento del código, en este caso, se ha optado por dejar los números directamente en el código debido a la naturaleza específica de las funciones y la claridad con la que se utilizan.

Aunque el uso de constantes simbólicas o enumeraciones podría proporcionar un mayor nivel de abstracción, se entiende que, en este contexto particular, los números utilizados para representar los pasos tienen una relación directa y comprensible con el flujo del proceso de votación. La introducción de variables adicionales o enumeraciones podría introducir una complejidad innecesaria en este caso específico.

Patrones GRASP

- ### Controlador (Controller):

El objeto `VotingKiosk` actúa como un controlador, ya que es responsable de coordinar el flujo del proceso de votación y manejar las interacciones con los servicios, el escrutinio y los dispositivos biométricos.

- ### Experto (Expert):

El método `grantExplicitConsent` en la clase `VotingKiosk` es responsable de otorgar el consentimiento explícito, y la clase `VotingKiosk` tiene acceso a la información necesaria para realizar esta tarea.

- ### Creador (Creator):

Los métodos `setHumanBiometricScanner` y `setPassportBiometricReader` en la clase `VotingKiosk` se utilizan para inyectar instancias de clases que implementan las interfaces `HumanBiometricScanner` y `PassportBiometricReader`, respectivamente, actuando como un creador de instancias.

- ### Indirecto (Indirection):

El uso de las interfaces `HumanBiometricScanner` y `PassportBiometricReader` proporciona una capa de indirección entre `VotingKiosk` y las implementaciones específicas (`StubHumanBiometricScanner` y `StubPassportBiometricReader`), reduciendo la dependencia directa.

- ### Variación Protegida (Protected Variations):

La interfaz `BiometricData` proporciona una capa de protección para la variación en la implementación de datos biométricos, ya que `VotingKiosk` interactúa con instancias de esta interfaz en lugar de detalles específicos de implementación.

- Asociación de Contenedor (Container):

La clase `VotingKiosk` actúa como un contenedor que almacena y coordina varios servicios, dispositivos biométricos y datos relacionados con el proceso de votación.

Principios SOLID

En nuestro código no se muestra una aplicación directa de todos los principios SOLID, pero se pueden identificar algunos de ellos:

- ### Principio de Segregación de Interfaces (ISP):

Las interfaces `HumanBiometricScanner` y `PassportBiometricReader` pueden considerarse de tamaño moderado, pero aún podrían dividirse en interfaces más específicas si es necesario para evitar que las clases implementen métodos que no utilizan.

- ### Principio de Inversión de Dependencias (DIP):

Se utilizan interfaces para la dependencia de los servicios y dispositivos biométricos (`HumanBiometricScanner` y `PassportBiometricReader`), lo que cumple con el DIP al depender de abstracciones en lugar de implementaciones concretas.

En resumen, mientras que algunos principios SOLID se aplican parcialmente en el código, hay oportunidades para mejorar la adherencia a estos principios mediante la refactorización y la reorganización del código para lograr una mayor cohesión y menor acoplamiento.

Enlace Al Repositorio De GitHub

https://github.com/k4rr3/AccessibleCitizenManagement