Franco Barlocco - Nicolas Lorenzo - Lucas Benitez

UT5TA2

Ejercicio 1)

Esta clase asume el anti patrón “The blob” ya que adquiere muchas responsabilidades generando que el código sea confuso a la hora de utilizarlo.

Para solucionar esto debemos separar las funcionalidades en diferentes clases por ejemplo, una clase con los métodos de gestión de tareas, otra con los métodos de gestión de usuarios, otra con actualizaciones del controller (add, delete, update), etc.

Ejercicio 2)

Esta clase asume el anti patrón “The blob” dado que debería separarse en varias clases dependiendo las responsabilidades, por ejemplo.

Deberíamos tener una clase para crear y borrar usuarios, otra clase para modificarlos, etc.

Ejercicio 3)

Posible godenHammer ya que utiliza una sola metodología para resolver el problema.

Se deberían implementar diferentes metodologías que se adecuen a cada problema.

Ejercicio 4)

Se implementa el anti patrón “Cut and Paste programming” ya que se crean dos métodos similares con funcionalidades diferentes.

Se debería de reutilizar el código de validateUserInput( ) para que funcione adecuadamente en validateUserPassword().

Ejercicio 5)

Anti patrón “Lava Flow” ya que el nombre de la función nos permite identificar que es código obsoleto.

Para esto debemos entender que hace el código para refactorizarlo si es que se utiliza, o en caso de ser necesario eliminarlo.

Franco Barlocco - Nicolas Lorenzo - Lucas Benitez

Ejercicio 6)

Se utiliza el anti patrón “Golden hammer” ya que se intenta generar una función forzada que no aborda de forma eficaz los problemas específicos, es decir, implementa métodos que realizan cosas diferentes a las que realmente quiere hacer.

Para solucionar esto debemos crear una función que haga su trabajo de forma correcta.

Ejercicio 7)

“Golden Hammer” ya que se utiliza una librería específica para hacer todas las funcionalidades del programa.

Para solucionar deberíamos evaluar que librería utilizar en cada funcionalidad, método o clase.

Ejercicio 8)

“The blob” ya que asume demasiadas responsabilidades sin tener en cuenta que cada clase debería tener una única responsabilidad para cambiar.

Para solucionar esto debemos genera diferentes clases para cada tarea diferente.

Ejercicio 9)

“Spaghetti code” ya que es código desordenado y difícil de leer con muchas dependencias y una estructura que no es clara.

Para solucionarlo debemos refactorizar el código de una forma clara y sencilla de leer.

Ejercicio 10)

“Tester driven development” ya que no se integraron practicas de calidad desde el inicio del desarrollo, obligando a realizar cambios luego de desarrollado el sistema. Esto genera ciclos de desarrollo más largos y costosos.

Para solucionar esto debemos integrar pruebas de testing desde el inicio del desarrollo.