Antipatrones

▼ The Blob

Ocurre cuando una clase o componente central en un sistema se convierte en una entidad masiva que contiene la mayor parte de la lógica y la funcionalidad del sistema.

Esta clase se vuelve tan grande y compleja que se convierte en un "monstruo" que absorbe y controla todas las operaciones del sistema.

▼ Ejemplos y correcciones

▼ Problema: TODOController

```
public class TODOController
{
    private List<Todo> todos;

    public TODOController()
    {
        this.todos = new List<Todo>();
    }

    public void Add(Todo todo)
    {
        todos.Add(todo);
    }

    public void Delete(int id)
    {
        Todo todo = todos.Find(t => t.Id == id);
        if (todo != null)
            todos.Remove(todo);
    }

    public void Update(Todo todo)
    {
        Todo oldTodo = todos.Find(t => t.Id == todo.Id);
        if (oldTodo != null)
        {
            oldTodo.Title = todo.Title;
            oldTodo.Description = todo.Description;
            oldTodo.Completed = todo.Completed;
        }
    }

    // Aquí hay más métodos relacionados a la gestión de tareas (TODOs)
    // ...
    // Y después, también hay métodos de gestión de usuarios.
    // ...
    // Y aún más, métodos para la gestión de permisos.
    // ...
}
```

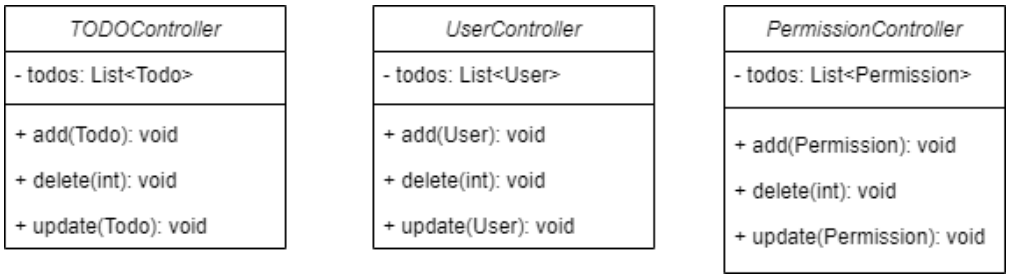
La clase `TODOController` es un claro ejemplo del antipatron *The blob* porque la misma posee muchas funcionalidades y centraliza mucha

logica. Debido a esto se puede considerar tambien que se rompe con el principio SRP.

Corrección:

Separar la logica de los usuarios y permisos en clases distintas. Eventualmente, si la logica de cualquiera de estas clases se vuelve muy extensa y/o compleja se puede considerar separarlas tambien en otras clases para mayor entendimiento.

▼ UML

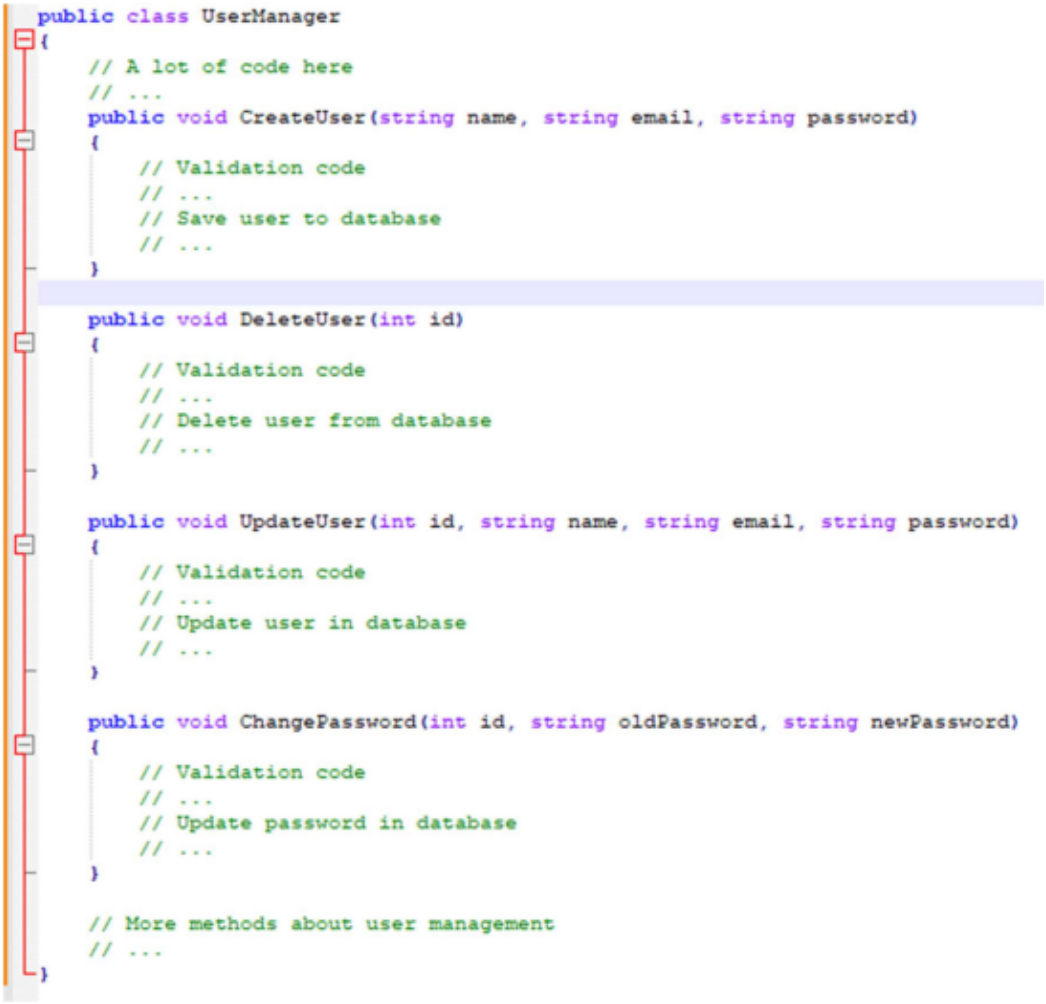


▼ Código

```
public class TODOController
{
  private List<Todo> todos;

  public TODOController()
  {
    this.todos = new List<Todo>();
  }

  public void add(Todo todo)
  {
    todos.Add(todo);
  }

  public void Delete(int id)
  {
    Todo todo = todos.Find(t => t.Id == id);
    if (todo != null)
      todos.Remove(todo);
  }

  public void Update(Todo todo)
  {
    Todo oldTodo = todos.Find(t => t.Id == todo.id);
    if (oldTodo != null)
    {
      oldTodo.Title = todo.Title;
      oldTodo.Description = todo.Description;
      oldTodo.Completed= todo.Completed;
```

```
    }
  }
}

public class UserController
{
  private List<User> users;

  public UserController()
  {
    this.users = new List<User>();
  }

  //Logica de usuarios

}

public class PermissionController
{
  private List<Permission> permissions;

  public PermissionController()
  {
    this.permissions = new List<Permission>();
  }

  //Logica de permisos
}
```

▼ Problema: UserManager



```
public class UserManager
{
    // A lot of code here
    // ...
    public void CreateUser(string name, string email, string password)
    {
        // Validation code
        // ...
        // Save user to database
        // ...
    }

    public void DeleteUser(int id)
    {
        // Validation code
        // ...
        // Delete user from database
        // ...
    }

    public void UpdateUser(int id, string name, string email, string password)
    {
        // Validation code
        // ...
        // Update user in database
        // ...
    }

    public void ChangePassword(int id, string oldPassword, string newPassword)
    {
        // Validation code
        // ...
        // Update password in database
        // ...
    }

    // More methods about user management
    // ...
}
```

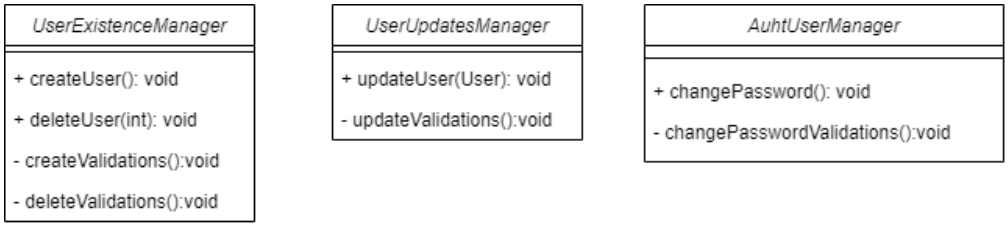
UT5-TA3

La clase `UserManager` posee demasiadas responsabilidades y funcionalidades, el centrar todo la lógica de manejo de usuarios en una sola clases puede tener consideraciones negativas en la mantenibilidad del código y la extension del mismo programa.

Corrección:

Se podría dividir funcionalidades del manejo de los usuarios en varias clases seleccionando un criterio adecuado para lograr cumplir con los principios pertinentes, como el SRP. Un posible solución a esto es separar lógica de creación y eliminación de usuarios en una clase, la de actualizaciones en otra, y la de cambios de contraseñas en otra.

▼ UML



▼ Código

▼ Lava Flow

Se refiere a la presencia de código obsoleto, inútil o no utilizado en un sistema de software. El código obsoleto es similar a la lava que fluye lentamente, ocupando espacio y consumiendo recursos sin aportar ningún valor real al sistema.

▼ Golden Hammer

Se refiere al fenómeno de utilizar una única herramienta o tecnología para resolver todos los problemas de desarrollo de software, sin considerar las necesidades y características específicas del proyecto. Este antipatrón se asemeja a usar un martillo dorado para todo, incluso cuando otras herramientas podrían ser más apropiadas.

▼ Tester driven development

Nuevos requerimientos son definidos en la fase de pruebas.

lE QA define nuevos requerimientos según la fase de pruebas.

▼ Spaghetti Code

Es un término que se utiliza para describir un código fuente de software que es confuso, desorganizado y difícil de entender.

También en OOP cuando los objetos son demasiado largos y desordenados.

▼ Cut−and−Paste Programming

Se refiere a un enfoque de desarrollo de software en el que se copian y pegan fragmentos de código sin un entendimiento completo de su funcionalidad o sin considerar la modularidad y la reutilización del código.

▼ Monster commit

Se refiere a una mala práctica en el desarrollo de software que consiste en realizar un solo y enorme commit que contiene múltiples cambios, características o funcionalidades en un solo conjunto. En lugar de dividir los cambios en commits más pequeños y lógicos, se agrupan todos los cambios en un solo commit masivo.