Campos dinámicos

Contexto

Como los campos del cuestionario son indicados como “dinámicos” desde la consigna, decidimos usar un diccionario para representarlos.

Opciones:

1) Los campos extras están en una nueva clase Formulario:
Tendría dos diccionarios de campos uno con los campos obligatorios y otro con los campos dinámicos.

Si se quiere eliminar un campo del formulario vamos a tener un método que sea *quitarCampo*().
Si se quiere agregar uno nuevo se usa el método *agregarCampo()* que crea un nuevo campo y se agrega a lista de campos.

2) No creamos una clase formulario, el colaborador tiene un diccionario con los campos extras.

Cuando se cree el formulario para que el colaborador complete, se verificará en *camposExtra* que campos poner en el cuestionario. Además de incluir los atributos propios de la clase como nombre, apellido, etc. estos serán obligatorios.

El jurídico y la persona normal tendrían campos extra distintos.

Elección: camposExtra en persona

Nos basamos en la cualidad de diseño “simplicidad” a la hora de elegir la segunda opción. Creando una nueva clase que no es del todo necesaria suma complejidad accidental a nuestra solución.

Quien se encarga de dar de alta, baja y modificar heladeras

Contexto:

Cuando nos enfrentamos al requerimiento que especifica que “se debe permitir el alta, baja y modificación de heladeras”, nos surge la siguiente incógnita: quién es el actor que lleva a cabo este caso de uso. Pensamos cuatro alternativas:

Opciones:

1) Cualquier colaborador puede modificar cualquier heladera: si todos los otros requerimientos lo hace un colaborador, entonces quizás podemos suponer que este caso de uso también.

2) Asumimos que no se da el caso de una heladera sin colaborador jurídico asociado, el caso de uso lo llevaría a cabo el colaborador jurídico a cargo de la heladera.

3) Distinguimos los casos en los que una heladera pertenece a un colaborador jurídico (y serían administradas por él) de las posibles heladeras que la ONG decida poner en funcionamiento por su cuenta.

4) La ONG se encarga totalmente de la administración de las heladeras en la solución.

Opiniones de cada alternativa

1) Luego de analizar la primera opción, concluímos que su implementación sería sumamente insegura. El hecho de que cualquier colaborador pueda modificar todas las heladeras presenta riesgos para la integridad y potencialmente la confidencialidad de la solución.

2) Examinando la segunda opción, nos alarma el hecho de que una entidad externa pueda acceder directamente a nuestro sistema, esto nos daría un alto grado de dependencia con la organización y es también un riesgo para la seguridad del sistema. Sumado a eso, implementar esta alternativa implicaría restringir el dominio de la problemática del cliente (asumir que una heladera siempre está asociada a un colaborador jurídico).

3) Además de la dependencia con una organización externa, analizada también en la alternativa dos, esta elección no es del todo consistente y suma complejidad innecesaria. Por otro lado, su implementación sí nos permitiría poner en funcionamiento una heladera, más allá de si está asociada a un colaborador jurídico o no.

4) La cuarta posibilidad es la única que no atenta contra la seguridad de las formas descriptas anteriormente. Tampoco limita el problema ni suma complejidad innecesaria.

Elección: La ONG se encarga de administrar las heladeras.

Herencia en clase Colaborador

Contexto:

Tenemos que modelar dos tipos de colaboradores: los jurídicos y los humanos. Viendo que tienen en común planteamos la posibilidad de que hereden de una clase Colaborador.

Opciones

1) Existe una clase colaborador de la que heredan la clase colaborador jurídico y el colaborador humano.

2) La clase colaborador jurídico y la clase colaborador humano son clases separadas que no heredan de una clase padre.

Elección

En este caso, fuimos por la alternativa número uno dado que maximiza la extensibilidad y reduce la redundancia. Esta abstracción, además, se asemeja al modelo presentado en la consigna.

~~~~~~~~~~~ o ~~~~~~~~~~~

PasswordValidator

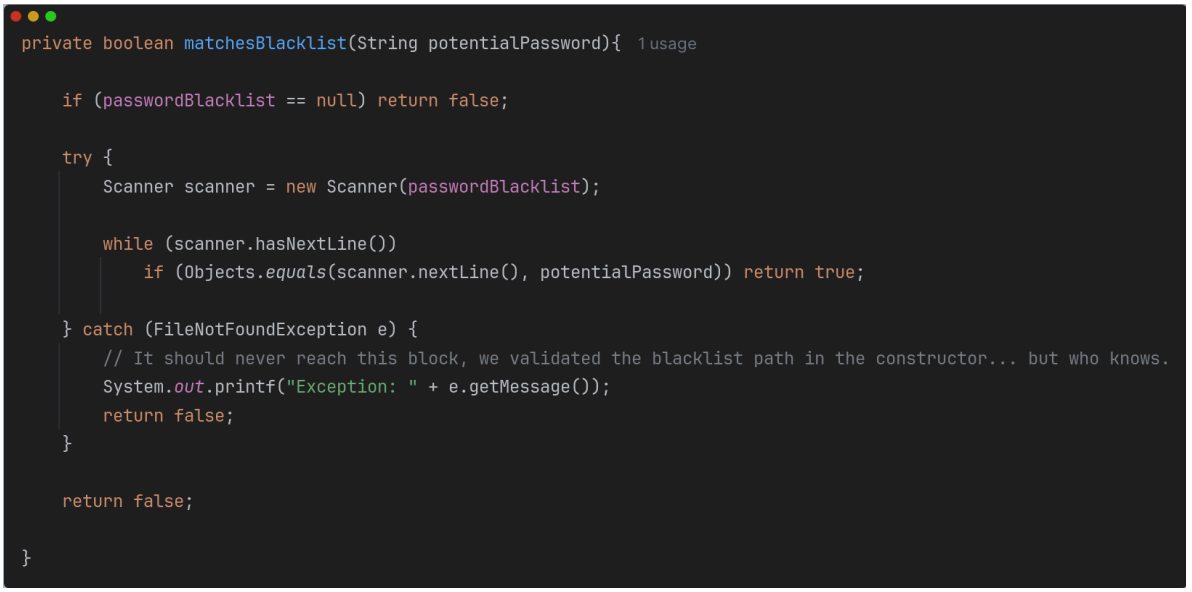
Implementando el validador de contraseñas como una clase separada ganamos reusabilidad, testeabilidad y mantenibilidad. Este componente de nuestro sistema presenta una alta cohesión dado que tiene una única responsabilidad y es: validar que una contraseña sea lo suficientemente segura.

Constructor

```java
PasswordValidator(String pathToPasswordBlacklist){  7 usages
    if (pathToPasswordBlacklist != null) {
        passwordBlacklist = new File(pathToPasswordBlacklist);
        if (!passwordBlacklist.isFile()) {
            System.out.println("Given path is not a valid file! blacklist test will not be conducted.");
            passwordBlacklist = null;
        }
    } else {
        System.out.println("Blacklist path is null, blacklist test will not be conducted!");
        passwordBlacklist = null;
    }
}
```

Pasando la blacklist como argumento al constructor, aumentamos la reusabilidad y, además, chequeando los distintos posibles escenarios e imprimiendo los mensajes de error adecuados, nuestro sistema se vuelve más robusto.
~~~~~~~~~~~

matchesBlacklist



```java
private boolean matchesBlacklist(String potentialPassword){  1 usage

    if (passwordBlacklist == null) return false;

    try {
        Scanner scanner = new Scanner(passwordBlacklist);

        while (scanner.hasNextLine())
            if (Objects.equals(scanner.nextLine(), potentialPassword)) return true;

    } catch (FileNotFoundException e) {
        // It should never reach this block, we validated the blacklist path in the constructor... but who knows.
        System.out.printf("Exception: " + e.getMessage());
        return false;
    }

    return false;

}
```

En este caso, implementando un try-catch la robustez del sistema incrementa.
Además implementamos diversos tests para validar el correcto funcionamiento del sistema.