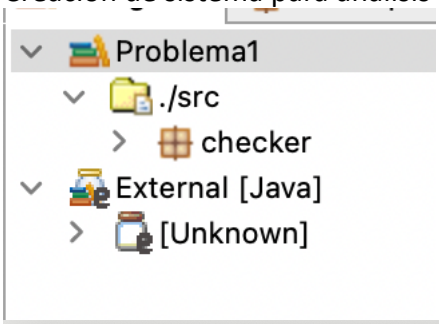
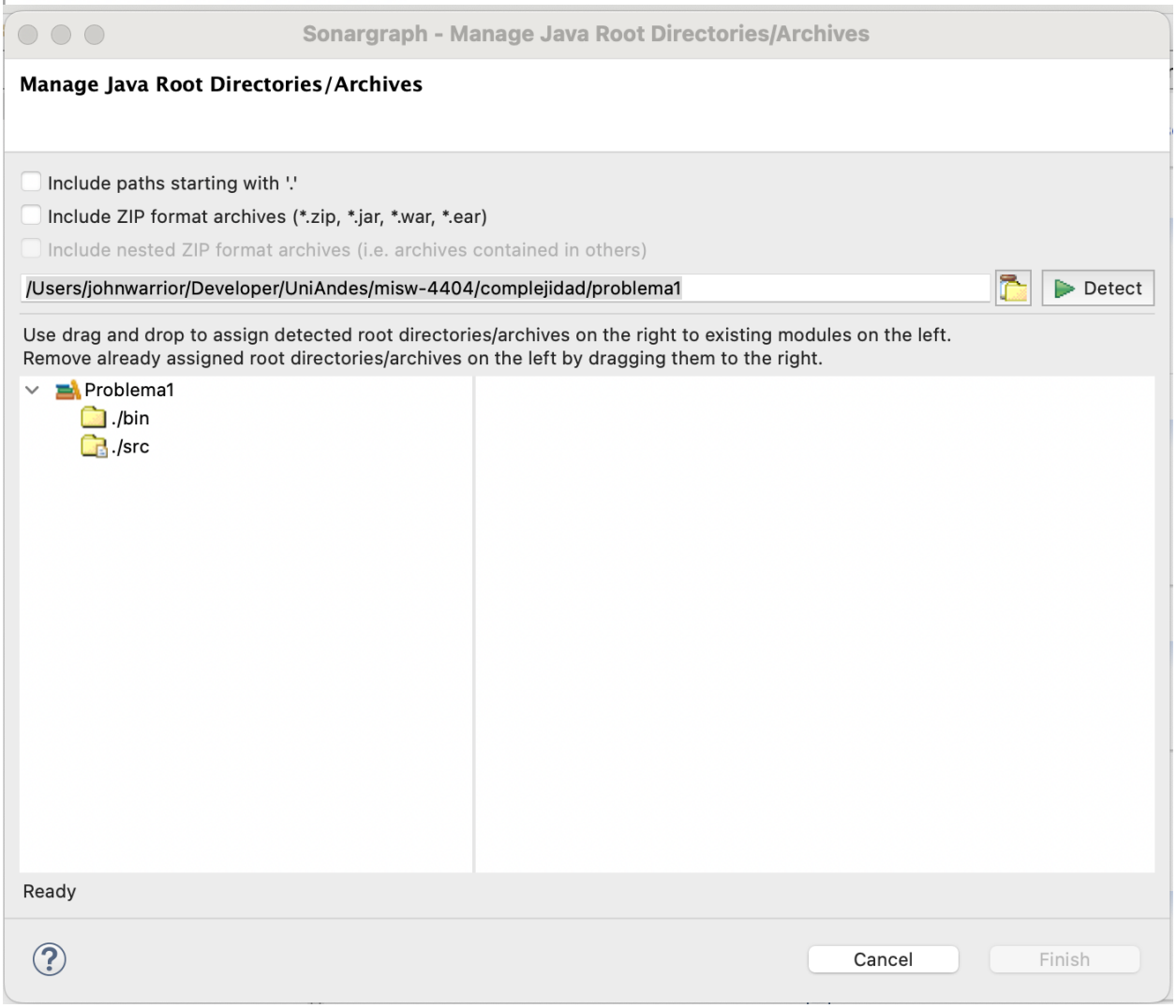
Complejidad

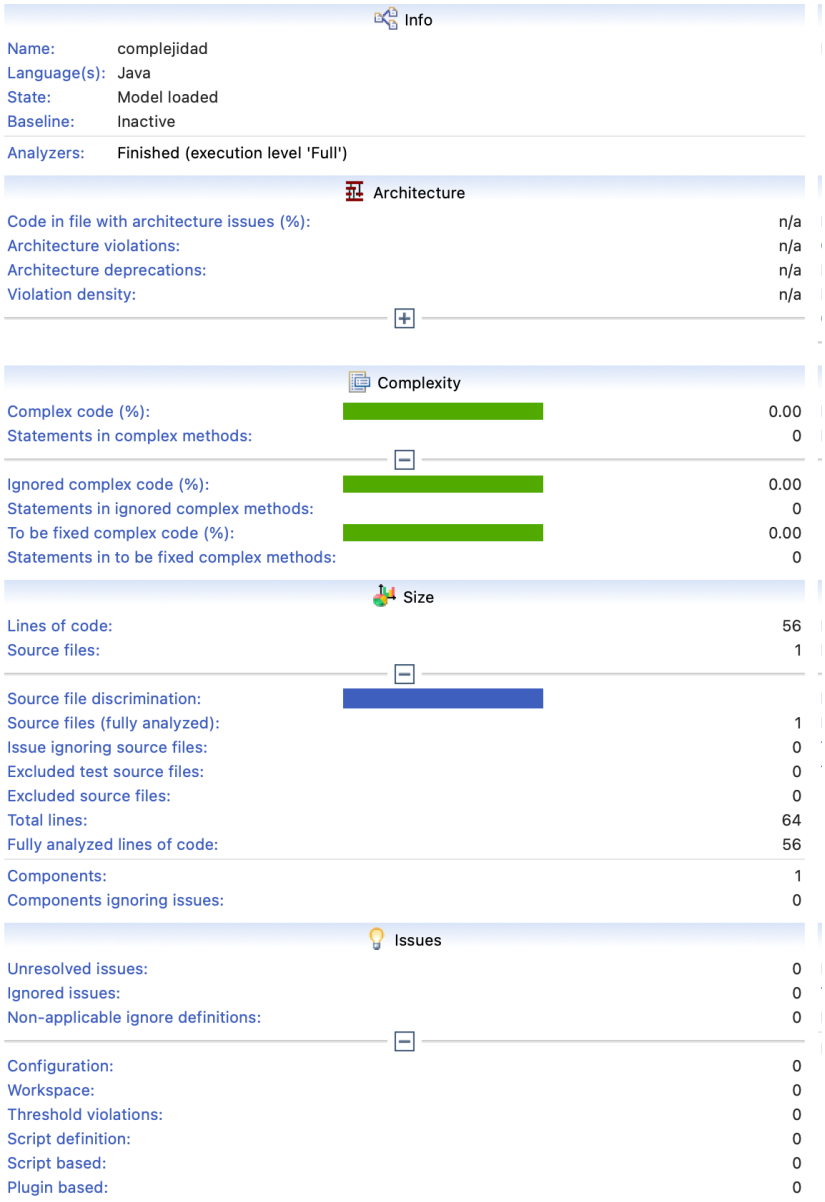
Problema 1

Creación de sistema para análisis en SonarGraph:





Análisis de complejidad del proyecto



Info

Name:	complejidad
Language(s):	Java
State:	Model loaded
Baseline:	Inactive
Analyzers:	Finished (execution level 'Full')

Architecture

Code in file with architecture issues (%):	n/a
Architecture violations:	n/a
Architecture deprecations:	n/a
Violation density:	n/a

Complexity

Complex code (%):	0.00
Statements in complex methods:	0
Ignored complex code (%):	0.00
Statements in ignored complex methods:	0
To be fixed complex code (%):	0.00
Statements in to be fixed complex methods:	0

Size

Lines of code:	56
Source files:	1
Source file discrimination:	
Source files (fully analyzed):	1
Issue ignoring source files:	0
Excluded test source files:	0
Excluded source files:	0
Total lines:	64
Fully analyzed lines of code:	56
Components:	1
Components ignoring issues:	0

Issues

Unresolved issues:	0
Ignored issues:	0
Non-applicable ignore definitions:	0
Configuration:	0
Workspace:	0
Threshold violations:	0
Script definition:	0
Script based:	0
Plugin based:	0

Aunque el análisis realizado por Sonargraph no reporta datos de la complejidad del programa realizando un análisis a nivel de código encontramos que la mayor complejidad se concentra en la función ‘checkPasswordStrenght’

```java
public String checkPasswordStrenght(char[] pass) {
    if(pass.length < 8)
        return "La contraseña es muy corta";

    boolean hasUppercase = false;
    boolean hasLowercase = false;
    boolean hasNumber = false;
    boolean hasSymbol = false;
    char[] specialChars = "!@#$%^&*()_+-=,./<>?;:[]{}|".toCharArray();

    for(char s : pass) {
        if(Character.isUpperCase(s)) {
            hasUppercase = true;
        } else if(!Character.isUpperCase(s)) {
            hasLowercase = true;
        } else if(Character.isDigit(s)) {
            hasNumber = true;
        } else {
            for( int i=0; i<specialChars.length; i++) {
                if(specialChars[i] == s) {
                    hasSymbol = true;
                    break;
                }
            }
        }

        if(!hasUppercase) {
            return "La contraseña debe tener una mayúscula";
        }
        if(!hasLowercase) {
            return "La contraseña debe tener una minúscula";
        }
        if(!hasNumber) {
            return "La contraseña debe tener un número";
        }
        if(!hasSymbol) {
            return "La contraseña debe tener un símbolo especial como !@#$%^&*()_+-=,./<>?;:[]{}|";
        }
    }
    return "la contraseña es segura";
}
```

Realizando una revisión de los posibles caminos que podría tomar la aplicación dentro de esta función tendríamos dos focos para tratar de disminuir la complejidad del códigos:

- En términos de O(n), para el ‘for’ loop encargado de iterar por el string de la contraseña, dependiendo de la extensión de dicha cadena de caracteres, dependería el valor de la complejidad del loop.
- En términos de O(1), para las validaciones que se realizan sobre la contraseña. Pero al mismo tiempo la complejidad de dicho código al momento de ser comprendido por otra persona es algo alta, dando cabida a la propagación de errores en la aplicación al encontrarse anidados los condicionales.

Refactoring

En este caso las formas de atacar los dos focos de complejidad identificados en los problemas serían:

1. Cambiar del uso de un ‘for’ loop para hacer la validar carácter por carácter de la contraseña a hacer el manejo de las validaciones del cumplimiento de cada una de las reglas a partir de las funciones que maneja JAVA de forma nativa para dicho propósito.

```
public String checkPasswordStrenght(char[] pass) {
    if(pass.length < 8)
        return "La contraseña es muy corta";

    String specialChars = "!@#$%^&*()_+-=,./<>?;:[]{}|";

    boolean hasUppercase = IntStream.range(0, pass.length) IntStream
            .mapToObj(i -> pass[i]) Stream<Character>
            .anyMatch(Character::isUpperCase);
    boolean hasLowercase = IntStream.range(0, pass.length) IntStream
            .mapToObj(i -> pass[i]) Stream<Character>
            .anyMatch(Character::isLowerCase);
    boolean hasNumber = IntStream.range(0, pass.length) IntStream
            .mapToObj(i -> pass[i]) Stream<Character>
            .anyMatch(Character::isDigit);
    boolean hasSymbol = IntStream.range(0, pass.length) IntStream
            .mapToObj(i -> pass[i]) Stream<Character>
            .anyMatch(ch -> specialChars.indexOf(ch) >= 0);
```

2. Las validaciones realizadas sobre cada uno de los caracteres se están realizando dos veces, tanto para validar el cumplimiento de la regla, así como para validar el mensaje que se ha de mostrar al final de la validación. Realizando el primer cambio propuesto se estaría dando solución a ambos inconvenientes, reduciendo la complejidad estructural del código y a las validaciones innecesarias sobre la cadena de caracteres.

```
if (!hasUppercase) {
    return "La contraseña debe tener una mayúscula";
}
if (!hasLowercase) {
    return "La contraseña debe tener una minúscula";
}
if (!hasNumber) {
    return "La contraseña debe tener un número";
}
if (!hasSymbol) {
    return "La contraseña debe tener un símbolo especial como !@#$%^&*()_+-=,./<>?;:[]{}|";
}
return "la contraseña es segura";
```

Luego del refactoring se mantuvo la métrica de complejidad del proyecto baja. Mejorando la legibilidad del código y su complejidad estructural.