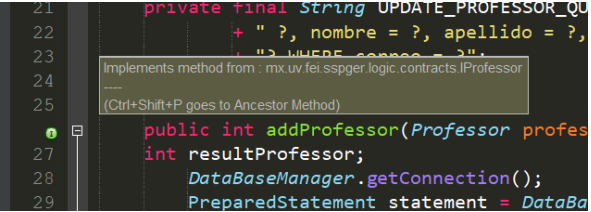
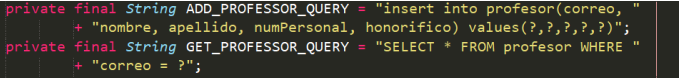
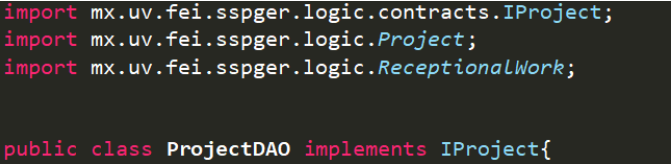


Criterio de evaluación	Descripción objetivo del criterio	Ejemplo de código tomado del proyecto final que no cumpla con el criterio	Ejemplo de código que cumpla con el criterio
Minimizar el ámbito de las variables	Declarando variables lo más tarde posible, cerca de su uso, para evitar confusiones y mejorar la claridad del código.	<pre>final int SUBSTRING_START = 1; private DeliverableFile deliverableFile;</pre> <p>@FXML deliverableFile está lejos de donde se usará</p>	<pre>private java.sql.Date getDeliverDate() { Date currentDate = new Date(); return new java.sql.Date(date: currentDate.getTime()); }</pre>
Minimizar la accesibilidad de clases y sus miembros	Evitar que se acceda a información que debería permanecer privada y que no debería ser modificada por otros módulos del programa.	No hay ejemplos	<pre>public class Professor extends User{ private String honorificTitle; private String personalNumber; private int id; public Professor(){ } public int getId() { return id; } }</pre>
Proveer retroalimentación del valor resultante de un método	Cuando una función regrese un valor deberá revisarse y manejar cualquier error si es que se ha producido. Algunas funciones pueden devolver errores a través de un mecanismo diferente y es por eso saber qué tipos devuelven y cómo manejarlos.	No hay ejemplos	<pre>result = statement.executeUpdate(); DataManager.closeConnection(); return result; }</pre>

Habilitar verificación de tipos en tiempo de compilación	Los compiladores de lenguajes de programación emiten varias advertencias cuando encuentran un código potencialmente defectuoso, hay que mantener estas advertencias habilitadas selectivamente y no ignorarlas.	No hay ejemplos	
Validación de constantes	Validar los valores constantes utilizados en el código para asegurarse de que están dentro de los límites aceptables y no pueden causar problemas en el programa. Esto puede incluir verificar que los valores sean del tipo y tamaño correctos, que no estén fuera de los límites definidos y que sean coherentes con el contexto de uso.	No hay ejemplos	
Evitar dependencia cíclica entre paquetes	Evitar que dos o más paquetes de código se dependan mutuamente, lo que puede causar errores difíciles de depurar y hacer que el código sea más difícil de mantener. Para evitar la dependencia cíclica, se pueden utilizar técnicas como la separación de responsabilidades, la creación de interfaces claras y el uso de inyección de dependencias.	No hay ejemplos	

Manejo de errores	Esto puede incluir capturar y registrar información detallada sobre los errores, proporcionar mensajes de error claros y útiles para los usuarios, y tomar medidas para evitar que los errores se propaguen y afecten el funcionamiento general del programa.	<pre>response = ResponseEntity(1, 2); } } catch (SQLException ex) { DataBaseManager.getConnection().rollback(); } finally { DataBaseManager.getConnection().close(); } return response;</pre> <p>El catch no maneja el error, solo enmascara posibles errores. Pues no se está poniendo un logger o algo donde se pueda encontrar información de la excepción.</p>	<pre> } } catch (SQLException ex){ Logger.getLogger(UserRegisterController.class.getName()).log(Level.SEVERE, null, ex); }</pre>
Diseño de interfaces	Asegurándose de que las interfaces estén diseñadas de manera clara y consistente, con una documentación adecuada y con la validación de los parámetros de entrada. También puede incluir la consideración de casos de error y excepciones que puedan ocurrir durante la interacción con la interfaz. Al diseñar interfaces defensivamente, se puede aumentar la confiabilidad y la robustez del código en general.	No hay ejemplos	<pre>public interface ISubmissionManager { int addSubmission(Submission submission, DeliverableFile file, int idAssignment) throws SQLException; int modifySubmission(Submission submission, int idSubmission) throws SQLException; }</pre>