**Object-Oriented Programming**

**HomeWork #20**

**DATE: 08/08/2023**

**NRC: 9652**

**TEAM 2**

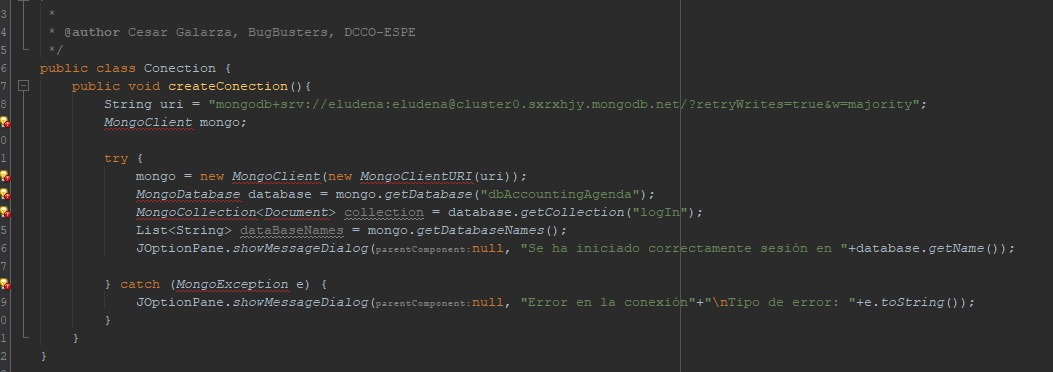
1. Galarza Luis 10 10
2. Ludeña Edison 10 10
3. Miranda Alison 10 10

INSPECTOR: TEAM #1

***SOLID PRINCIPLES THAT ARE USE***

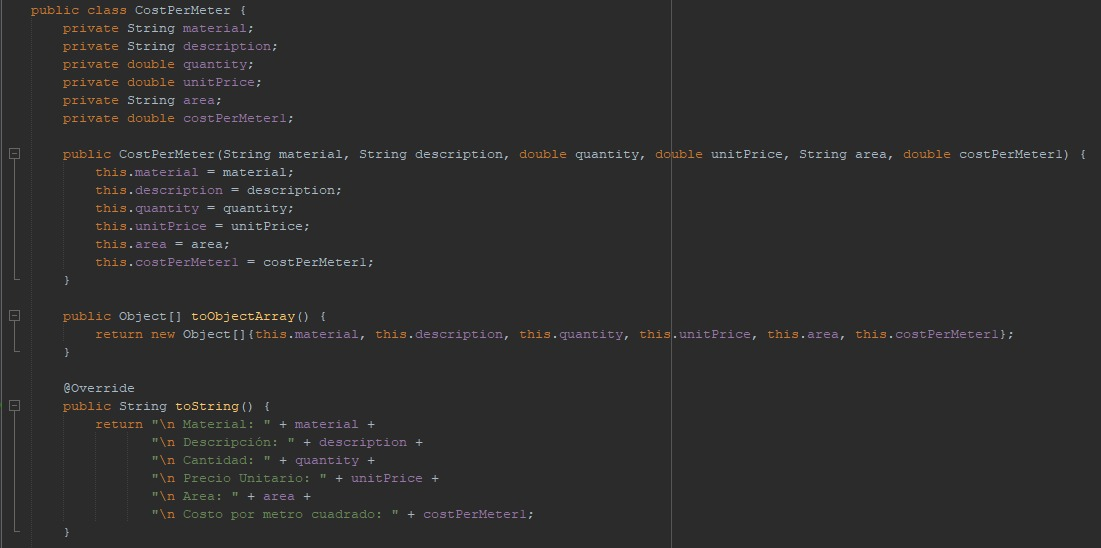
* **Class Connection**

Has single responsibility manages connection with MongoDB Database.



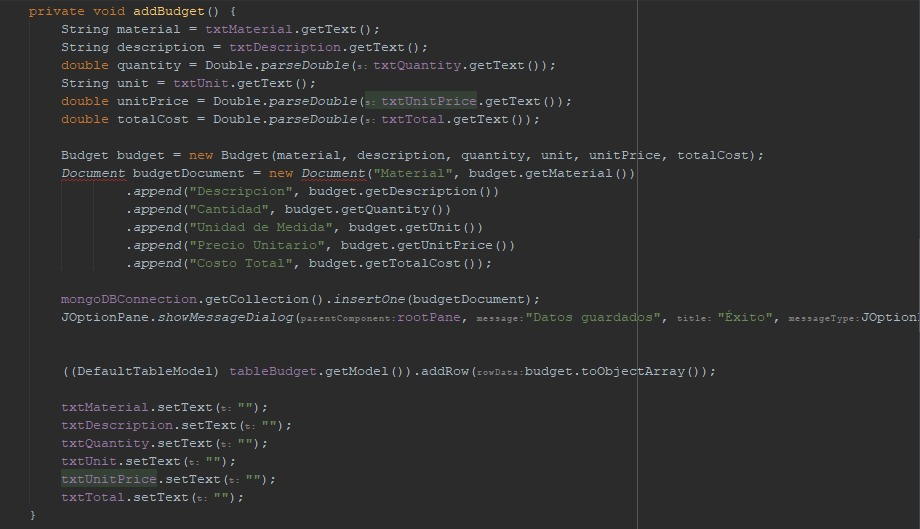
* **Class CostPerMeter**

Has single responsibility give information about the cost per square meter and give methods to access and edit that information.



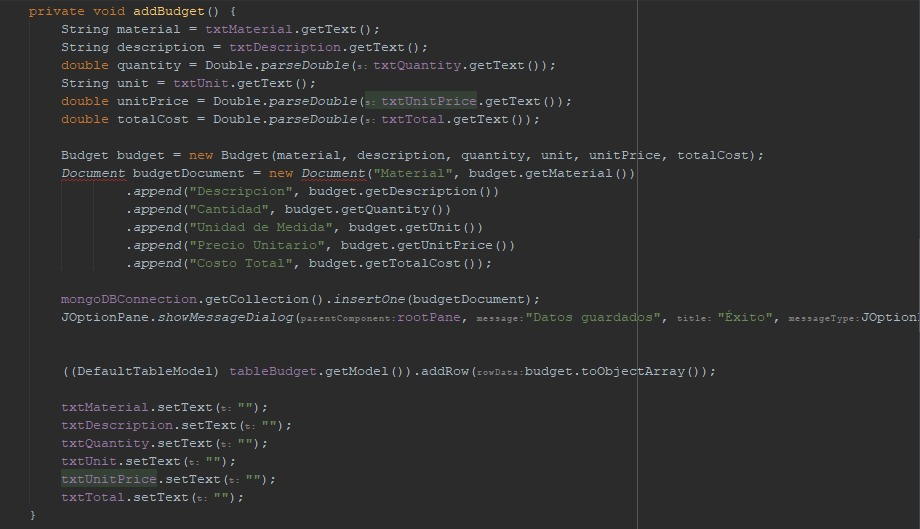
* **FrmBudget Class - Method addBudget()**

Has single responsibility, because make what needs to be made



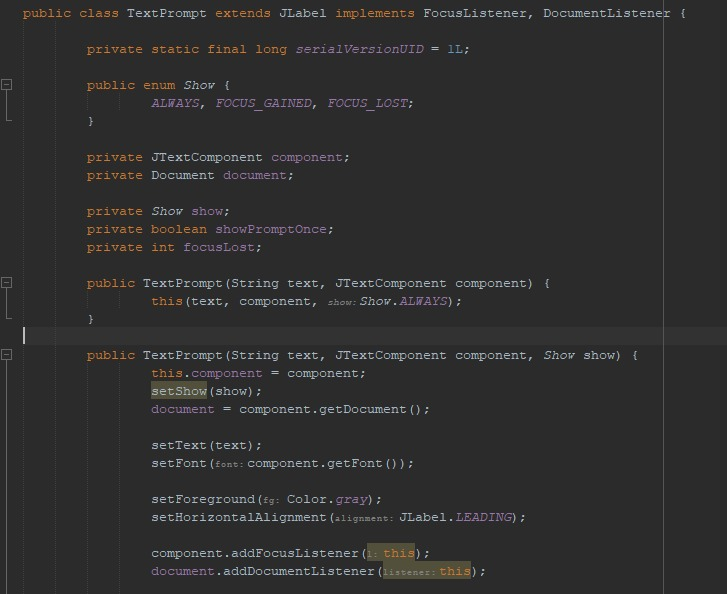
* **FrmBudget Class - Method addBudget()**

Has open/close, because can be used in other parts of the code, but can’t be modified.



* **Class TextPrompt**

Has single responsibility, because the main purpose is to manage and show the help text.



***SOLID PRINCIPLES THAT ARE NEED***

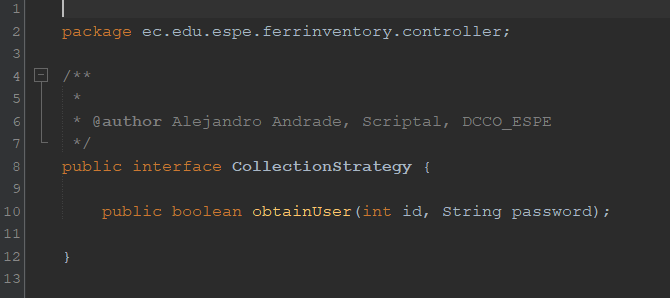
We can’t find more principles that the team can use.

**TEAM 3**

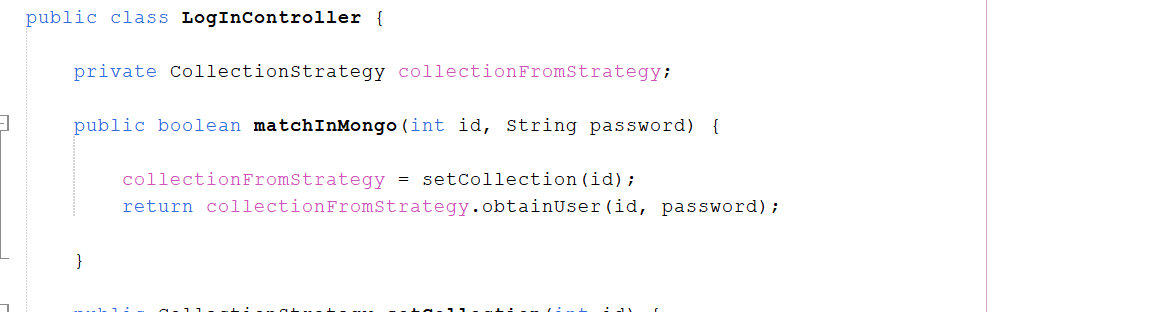
1. Moreno Paul 10 10
2. Paquel Juan 10 10
3. Ramos Javier 9 9
4. Rueda Juan 0 0

INSPECTOR: TEAM #2

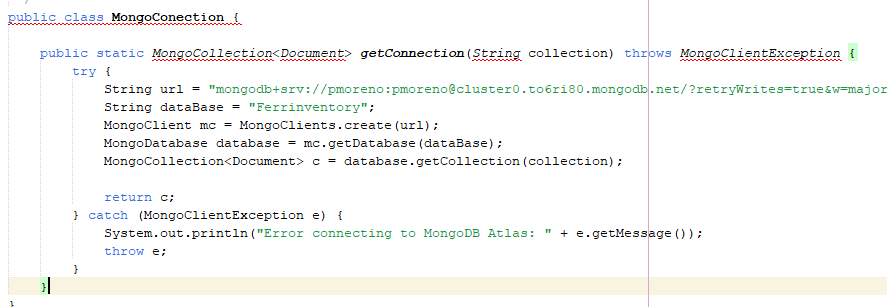
* **Class “CollectionStrategy”:** Use the Interface Segregation Principle (ISP) to design the specific interface at implementing classes' needs, with only one entry method.

****

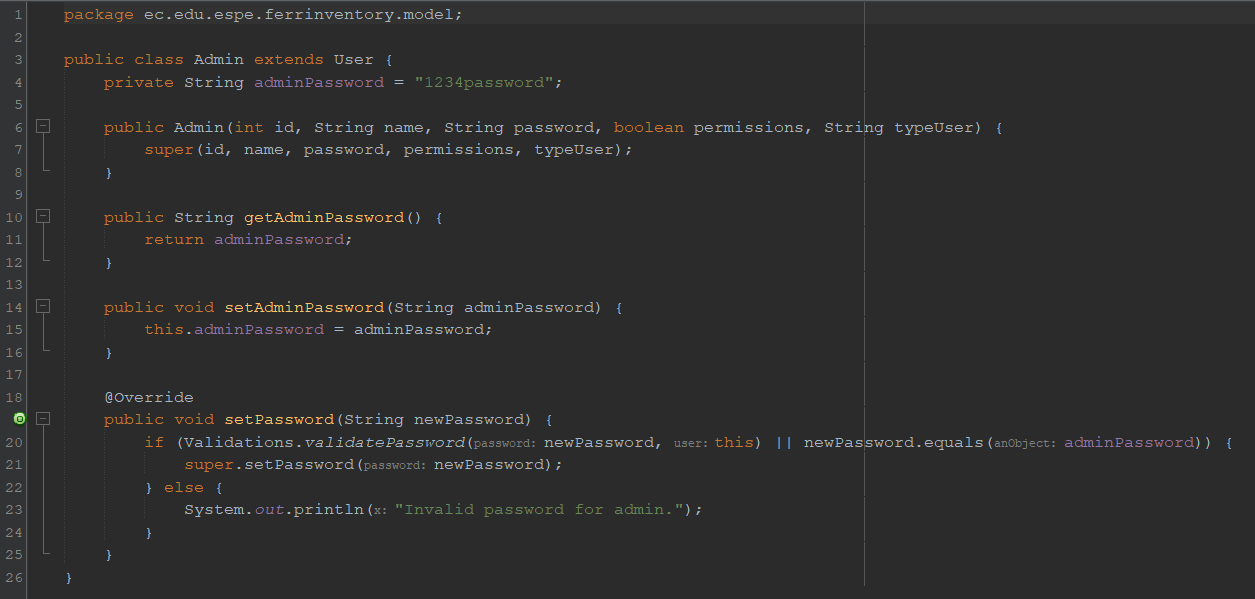
* **Class “LonInController”:** Has single responsibility principle, in the match with mongo



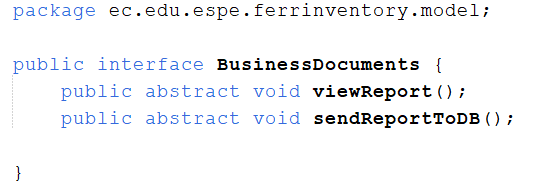
* **Class “MongoConection”:** Has single responsibility principle, only the connection



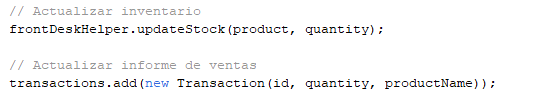
* **Class “Admin”:** Has Liskov Substitution Principle with this class can be substitutable with their superclass at the “Admin” child object.

****

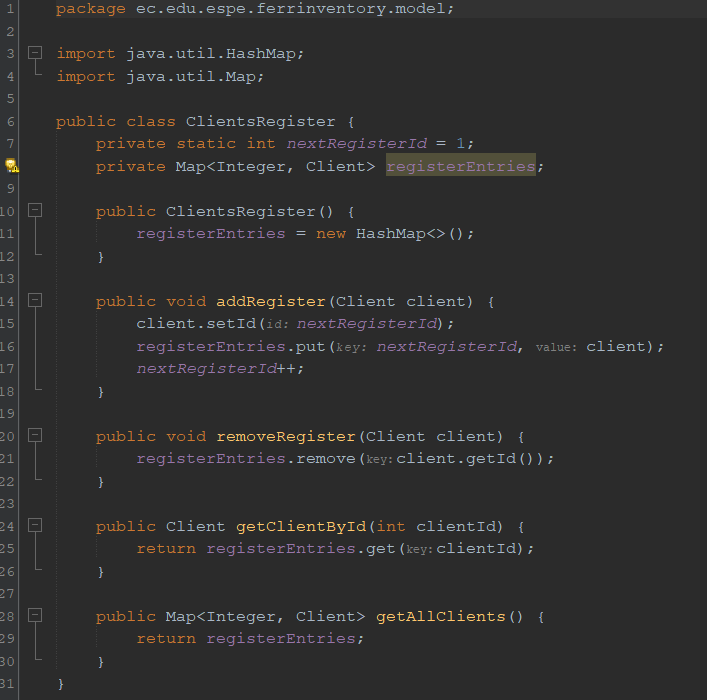
* **Class “BusinessDocuments”:** Use the Interface Segregation Principle (ISP) to design interfaces specific to implementing classes' needs, preventing unnecessary method requirements and enhancing code cohesion.



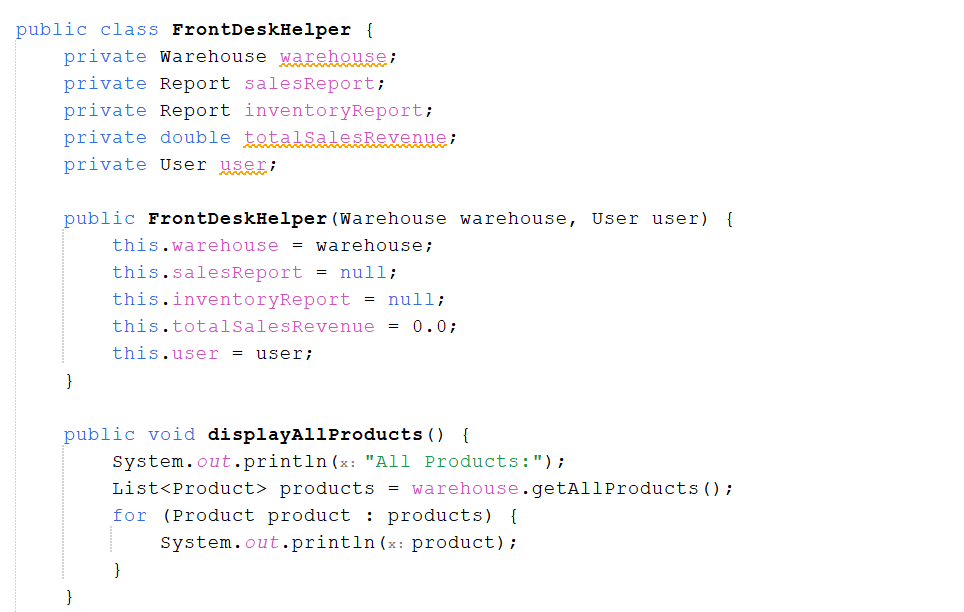
* **Class “Client”:** Has single responsibility, maybe it could be better at the direct dependency with FrontDeskHelper

****

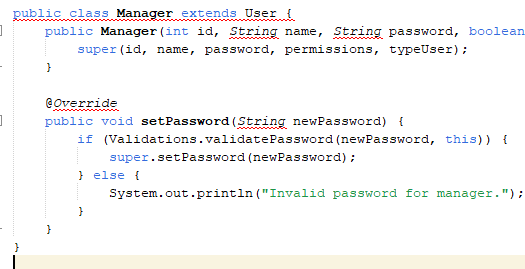
* **Class “ClientRegister”:** Use the dependency inversion principle with three methods that can make abstractions for each functionality.

****

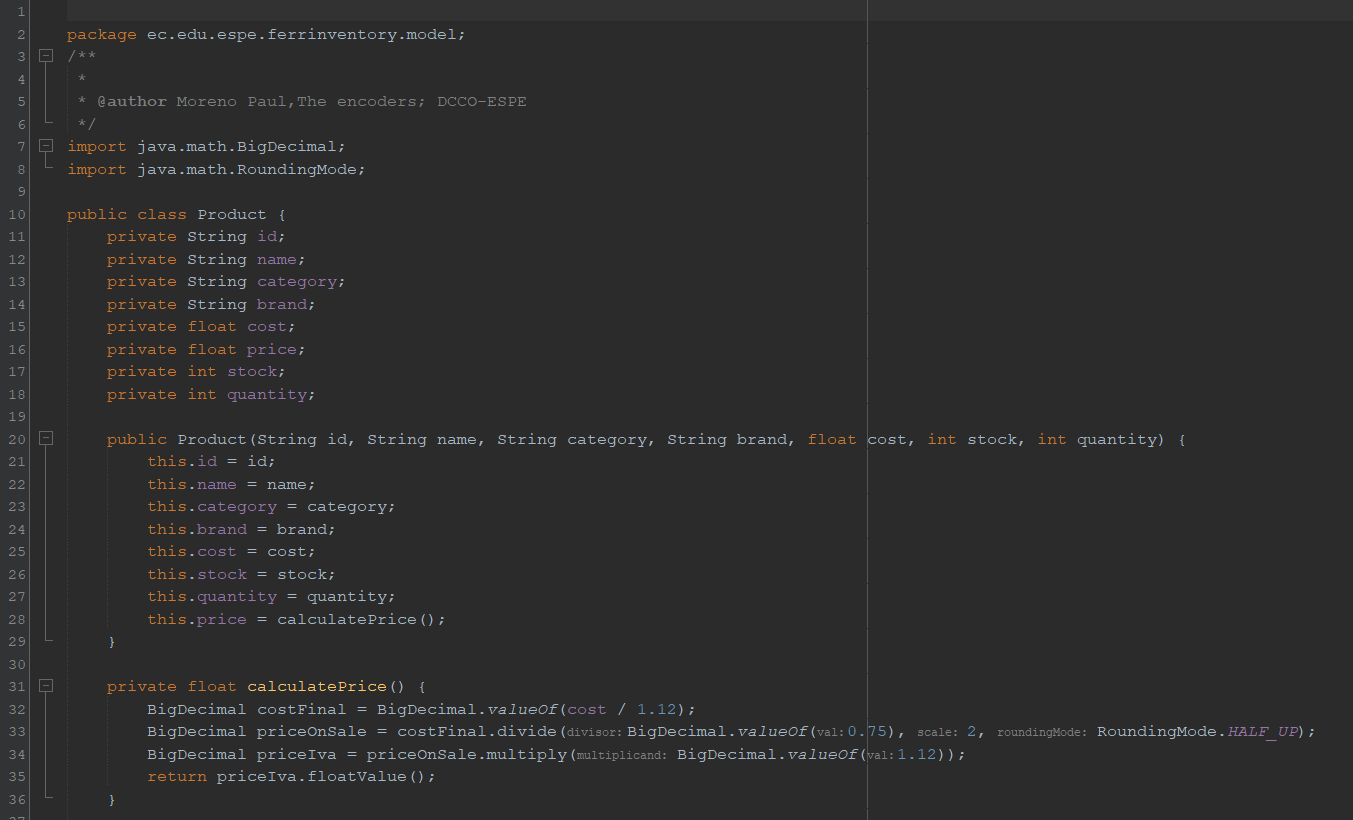
* **Class “FrontDesk”:** It does not have the principle of single responsibility since, being in the model package where the data is stored and sent, it also has methods that perform various actions.



* **Class “Manager”:** Has SRP and LSP, maybe its not to opened to modifications in the Manager attributes

****

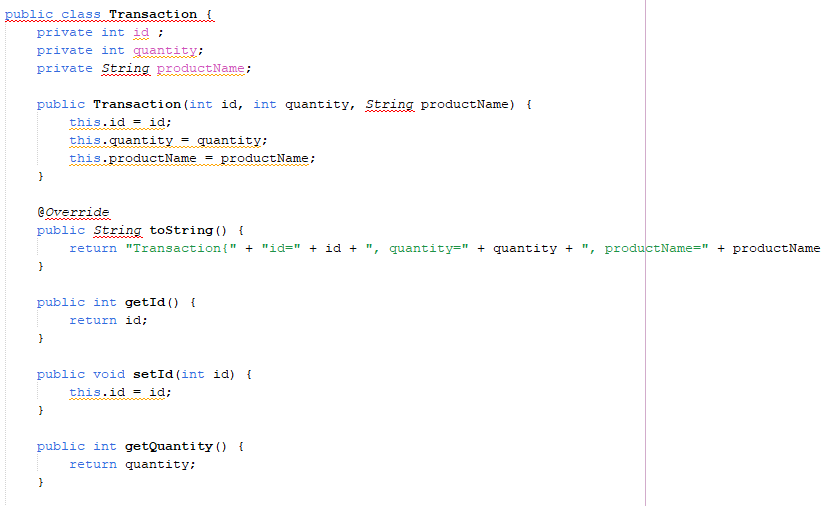
* **Class “Product”:** It does not have the principle of single responsibility since, being in the model package where the data is stored and sent.

****

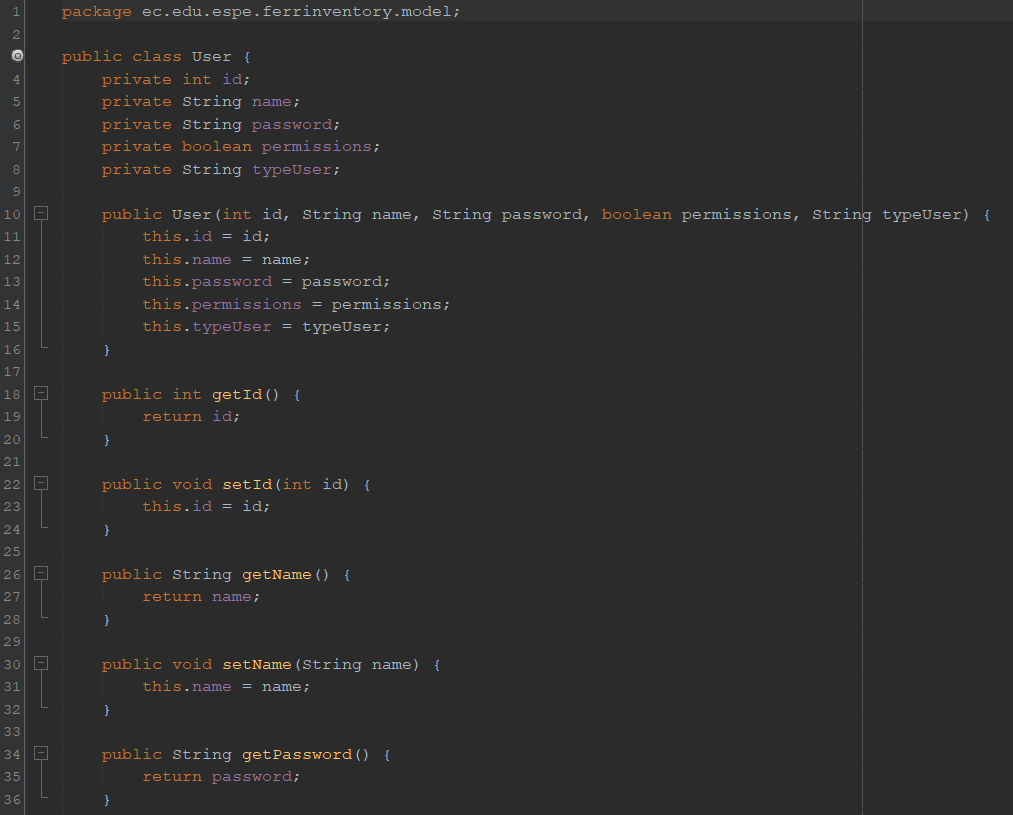
* **Class “Report”:** They use Liskov Substitution (LSP) to facilitate class sharing, but should consider enhancing SOLID principles by adopting Single Responsibility (SRP), split into report and business document for a clearer and easier to maintain design.



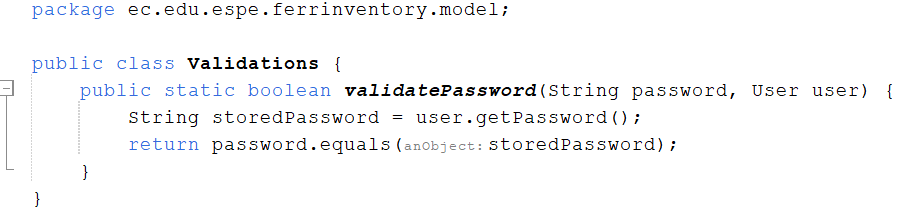
* **Class “Transaction”:** Ok

****

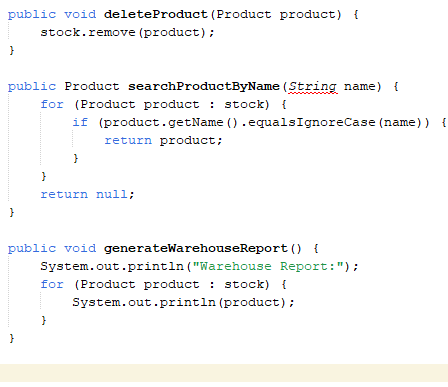
* **Class “User”:** It does not have the principle of single responsibility since, being in the model package and was made to instantiate an object, that’s the reason that was modificable.

****

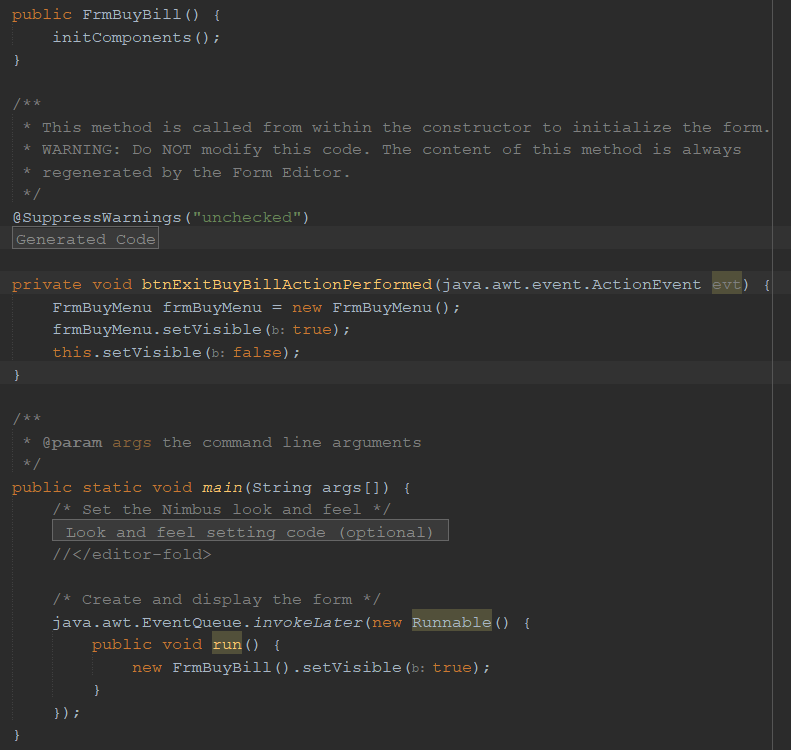
* **Class “Validations”:** The code does not use any SOLID principle at this time. It could apply the Single Responsibility Principle (SRP) by separating password validation and user property access into distinct classes for better organization and maintainability.



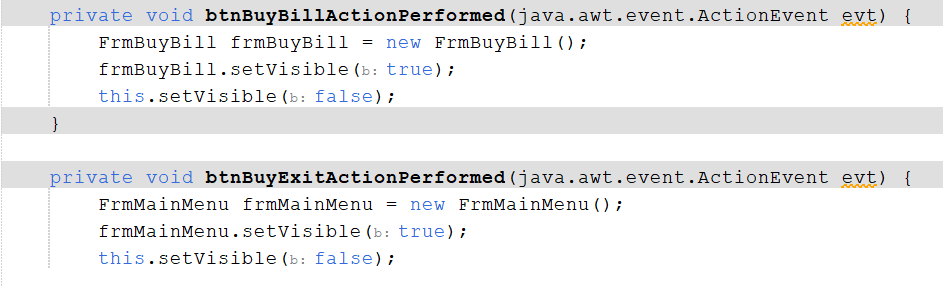
* **Class “WareHouse”:** Doesn’t have SRP



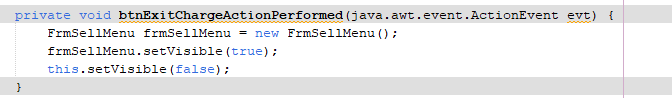
* **Class “FrmBuyBill”:** Has a single responsibility because only have one button to code an action event, so use OC to call “FrmBuyMenu”.

****

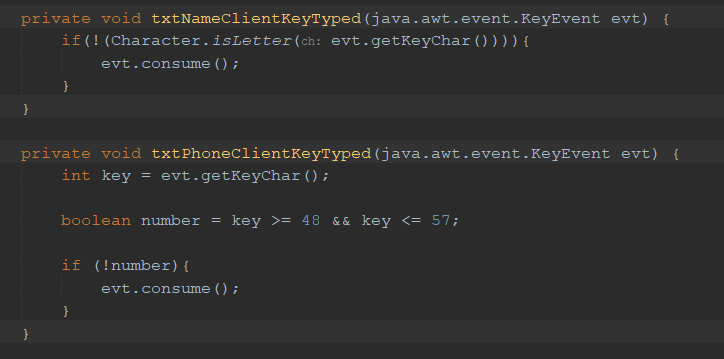
* **Class “FrmBuyMenu”:** Has a single responsibility because in each button there are only calls and no code or other complex processes.



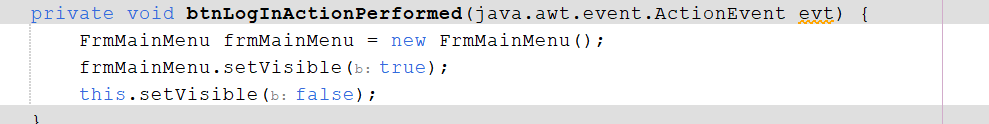
* **Class “FrmCharge”:** No functionality besides exit



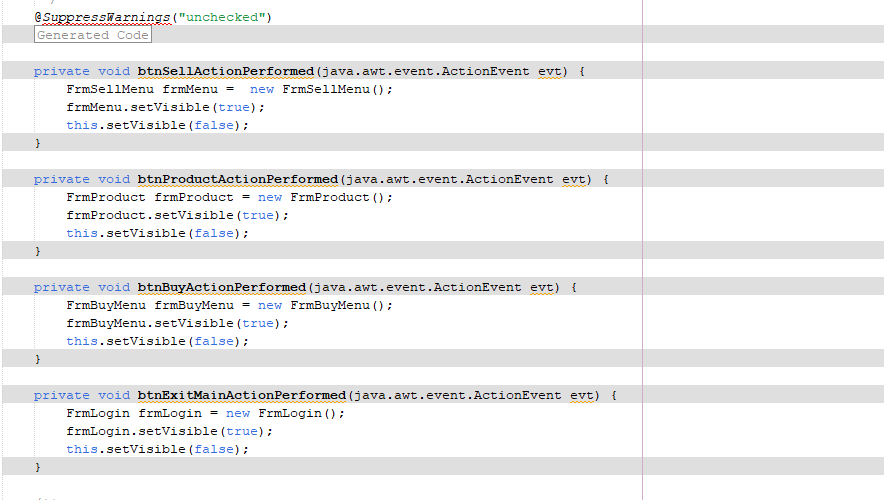
* **Class “FrmClient”:** Their could use the single responsibility method to not put processes on the buttons and make that process a method and call that function.

****

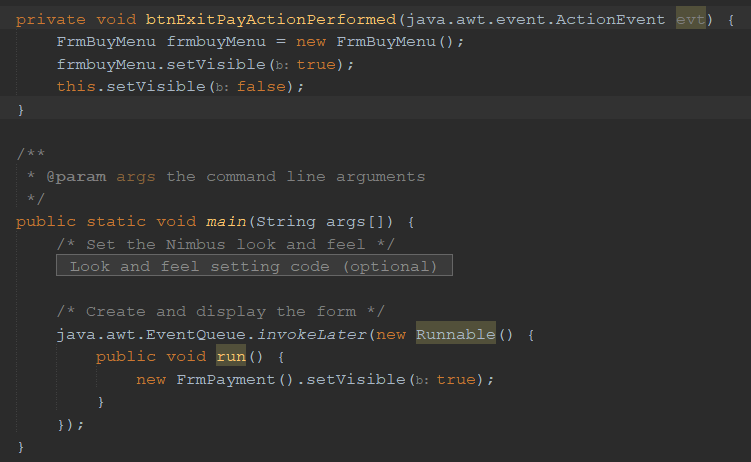
* **Class “FrmLogin”:** Has a single responsibility because in each button there are only calls and no code or other complex processes.



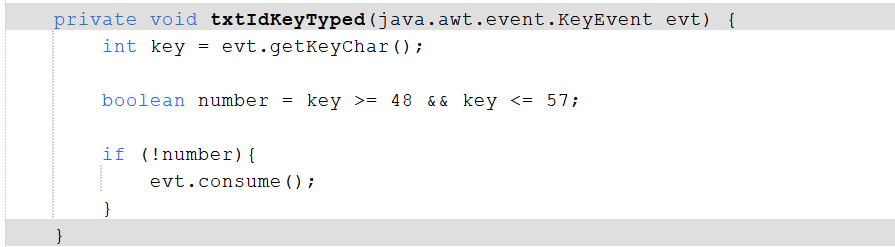
* **Class “FrmMainMenu”:** Ok

****

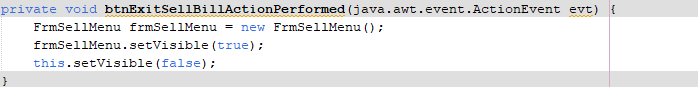
* **Class “FrmPayment”:** Has a single responsibility because only have one button to code an action event, so use OC to call “FrmBuyMenu” (duplicated code with “FrmBuyBill” for no more functions?).

****

* **Class “FrmProduct”:** Their could use the single responsibility method to not put processes on the buttons and make that process a method and call that function.



* **Class “FrmSellBill”:** No functionality besides exit

****

* **Class “FrmSellMenu”:** Ok

****

* **Class “FrmSuplier”:** You could use the single responsibility method to not put processes on the buttons and make that process a method and call that function.

