KHULNA UNIVERSITY

Computer Science and Engineering Discipline

Course No. : CSE-3106
Course Name : Software Development Laboratory

Code Review of The Software Project:

Inventory Management System

Project by:
- **Name:** Arnob Chakroborty
  Student ID: 210205
- **Name:** Samia Khanom Asha
  Student ID: 210222

Reviewed by:
- **Name:** Gayotry Gope Toma
  Student ID: 210212
- **Name:** Jannatul Ferdous Shova
  Student ID: 210228

Submitted to:
Dr. Amit Kumar Mondal
Associate Professor
Computer Science and Engineering
Khulna University, Khulna

Name of the Project:

Inventory Management System.

Introduction:

The provided code consists of multiple classes and functions for an Inventory Management System implemented using Tkinter. Below are some observations and recommendations for improvement:

Architecture Pattern:

The provided project follows a ***Layered Architecture Pattern***. A layered architecture pattern typically consists of distinct layers, each responsible for a specific aspect of the application's functionality. These layers include presentation (UI), business logic, and data access layers, among others. Each layer communicates with the layer directly below or above it, promoting separation of concerns and modularity.

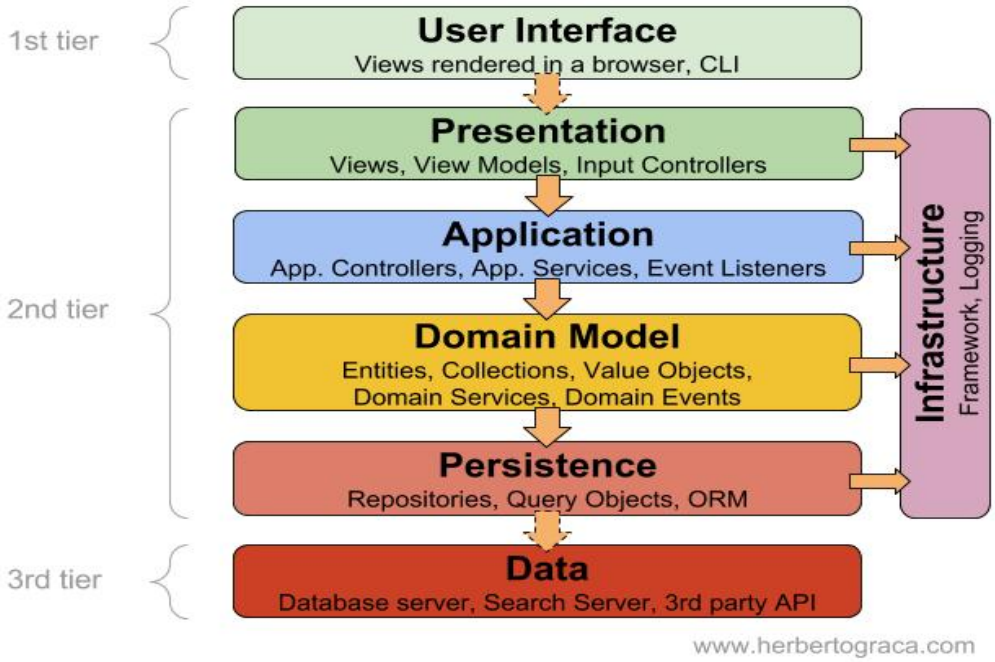

Figure: *Layered Architecture Pattern*

Code Smells:

- ***Code Duplication:***

There's some duplication of code across different classes, such as defining GUI elements like labels, entries, and buttons. Consider consolidating common GUI elements into a separate method or class to avoid redundancy.

- ***Responsibility Overload:***

Classes like InventoryManagementSystem, InventoryGUI, InventoryTrackerGUI, and InventoryManagementGUI handle both GUI construction and application logic. This violates the Single Responsibility Principle (SRP). Separation of concerns should be applied to make the codebase more modular and maintainable.

- ***Readability:***

The code lacks proper documentation (docstrings) explaining the purpose of classes and methods. Adding docstrings will improve code readability and maintainability. Variable names could be more descriptive, especially in classes like InventoryGUI, InventoryTrackerGUI, and InventoryManagementGUI

- ***User Feedback:***

User feedback messages displayed using messagebox are hard-coded strings. It's recommended to externalize these strings to improve internationalization and ease of maintenance.

- ***Error Handling:***

Error handling is minimal. It's recommended to handle potential exceptions more robustly, providing informative messages to the user. Consider using try-except blocks where file I/O operations are performed to handle potential errors gracefully.

List of Classes including Code Smells:

To identify classes that do not follow a standard coding convention in the given code base, we can look for inconsistencies such as naming conventions or structure. There are two classes in this code-base. Here's the analysis:

1. InventoryManagementSystem Class:

In the ***InventoryManagementSystem class***, the initial product data is hardcoded within the __init__ method. If this data were to change or need to be loaded from an external source, it would require modifying the class definition. It might be better to externalize this data to a configuration file or database for better flexibility.

```
86  class InventoryManagementSystem:
87      def __init__(self, master):
88          self.master = master
89          self.master.title("Inventory Management System")
90
91          self.products = {
92              "Mango": 0,
93              "Banana": 0,
94              "Pineapple": 0,
95              "Jackfruit": 0,
96              "Blueberry": 0,
97              "Raspberry": 0
98          }
99
100         self.create_widgets()
101
102     def create_widgets(self):
103         self.product_label = tk.Label(self.master, text="Select Product:")
104         self.product_label.grid(row=0, column=0, padx=10, pady=5)
105
106         self.product_var = tk.StringVar()
107         self.product_var.set("Mango")
108         self.product_dropdown = tk.OptionMenu(self.master, self.product_var, *self.products.keys())
109         self.product_dropdown.grid(row=0, column=1, padx=10, pady=5)
110
111         self.quantity_label = tk.Label(self.master, text="Enter Quantity:")
112         self.quantity_label.grid(row=1, column=0, padx=10, pady=5)
```

Figure: *InventoryManagementSystem class*

2. InventoryGUI Class:

Error handling is minimal in the ***InventoryGUI class***, there's no validation of input data (such as ensuring quantity is a valid number). Robust error handling could improve the resilience of the application.

Besides, this class mixes responsibilities. It handles both GUI creation and updating inventory. Ideally, it should follow the Single Responsibility Principle (SRP), where each class should have only one reason to change. Separating GUI creation from inventory updating logic would make the class cleaner and more maintainable.

```
139  class InventoryGUI:
140      def __init__(self, master):
141          self.master = master
142          master.title("Inventory Management System")
143
144          self.label = tk.Label(master, text="Update Product Information")
145          self.label.pack()
146
147          self.product_label = tk.Label(master, text="Select Product:")
148          self.product_label.pack()
149
150          self.product_options = ["Mango", "Banana", "Pineapple", "Jackfruit", "Blueberry", "Raspberry"]
151          self.product_var = tk.StringVar(master)
152          self.product_var.set(self.product_options[0])
153
154          self.product_menu = tk.OptionMenu(master, self.product_var, *self.product_options)
155          self.product_menu.pack()
156
157          self.quantity_label = tk.Label(master, text="Enter Quantity:")
158          self.quantity_label.pack()
159
160          self.quantity_entry = tk.Entry(master)
161          self.quantity_entry.pack()
162
163          self.update_button = tk.Button(master, text="Update", command=self.update_inventory)
164          self.update_button.pack()
```

Figure: *InventoryGUI class*

Specific Recommendations:

- ***Refactor Classes:***

Divide the functionality of the existing classes into smaller, more focused classes, adhering to the Single Responsibility Principle (SRP). Consider creating separate classes for GUI construction, user authentication, data manipulation, and inventory management.

- ***Modularize GUI Construction:***

Extract common GUI elements (labels, entries, buttons) into separate methods or classes to reduce duplication and improve maintainability.

- ***Improve Error Handling:***

Implement robust error handling mechanisms, including try-except blocks, to handle exceptions gracefully and provide informative error messages to users.

- ***Enhance Readability:***

Add docstrings to classes and methods explaining their purpose, inputs, and outputs.
Use meaningful variable names that convey the purpose of the variables.

- ***Separate Application Logic from GUI:***

Decouple application logic from GUI construction to improve code modularity and facilitate testing and maintenance.

- ***Consider Externalizing Constants:***

Externalize hard-coded strings (e.g., success and error messages) into constants or configuration files for easy modification and internationalization.

- ***Optimize Code Size:***

Refactor methods and classes exceeding the recommended size limits to improve code readability and maintainability.

Conclusion:

In conclusion, the provided code exhibits several code smells, including code duplication, responsibility overload, and readability issues. By refactoring the code following the recommended practices, such as modularization, error handling improvement, and separating concerns, the codebase can become more maintainable, readable, and scalable.