Grocery Inventory Manager

Author: James Kirk

Florida Atlantic University

Department of Electrical Engineering and Computer Science

COP 4331-003 Object Oriented Design & Programming

Platform: Java, SwingUI

1. Functional Requirements

1.1 Item Management

- The system shall allow the user to add new food items.
- Each food item shall have:
  - A name
  - A quantity
  - An optional expiration date
  - An optional category
  - An assigned storage location
- The system shall allow the user to edit existing food items.
- The system shall allow the user to delete food items.

1.2 Location Management

- The system shall allow users to create storage locations (pantry, fridge…)

- The system shall allow users to delete existing storage locations.
- The system shall display all food items assigned to a selected location.

1.3 Category Management

- The system shall allow users to create food categories.
- The system shall allow users to edit food categories.
- The system shall allow users to delete food categories.
- The system shall allow users to assign categories to food items.
- The system shall allow users to filter food items by category.

1.4 Expiration Tracking & Freshness Filtering

- The system shall allow users to optionally assign expiration dates to food items.
- The system shall determine the freshness status of an item based on the current date.
- The system shall allow users to filter items by:
  - Fresh
  - Expiring Soon
  - Expired

1.5 Sorting and Searching

- The system shall allow users to sort food items alphabetically by item name.
- The system shall allow users to sort food items alphabetically by category name.
- The system shall allow users to search for food items by name.

1.6 Grocery List Feature (Bonus Feature)

- The system shall allow users to create a grocery list.
- The system shall allow users to save the grocery list.
- The system shall allow users to print the grocery list.

2. Use Cases

UC1 – Add Location

Actor: User

Description: The user creates a new storage location using a GUI form.

Steps:

1. User clicks Locations button.
2. GUI opens a Locations Management window with a scrollable list.
3. User clicks Add Location button.
4. GUI displays a form with a textfield labeled Location Name, and Save and Cancel buttons.
5. User enters a non-empty location name.
6. User clicks Save.
7. System validates the entry.
8. System stores the new location in the database.
9. System updates the Locations list to show the new entry.

Variation 1 – Empty Name

1.1 User leaves the Location Name empty.

1.2 User clicks Save.

1.3 System displays popup: “Location name cannot be empty.”

1.4 User clicks OK, popup closes.

Variation 2 – Duplicate Name

2.1 System detects name already exists.

2.2 System shows error popup: “Location already exists.”

2.3 User clicks OK.

Variation 3 – Cancel

3.1 User clicks Cancel.

3.2 GUI closes form, no location added.

UC2 – Delete Location

Actor: User

Description: The user deletes a location. Deletion requires confirmation and deletes all items assigned to that location.

Steps:

1. User clicks Locations button.
2. System shows Locations Management window with list and Delete button.
3. User clicks a location in the list to select/highlight it.
4. User clicks Delete Location button.
5. System shows confirmation popup: “Are you sure you want to delete this location and all of its contents?”

6. User clicks Yes.
7. System deletes location and all items in it from the database.
8. System updates the list to remove the location.

Variation 1 – User Clicks No

1.1 User clicks No instead of Yes.

1.2 Popup closes and nothing is deleted.

Variation 2 – No Location Selected

2.1 User does not select a location.

2.2 User clicks Delete Location anyway.

2.3 System displays popup: “No location selected.”

2.4 User clicks Close.

UC3 – Select Location

Actor: User

Description: The user selects a location to view its food items.

Steps:

1. User opens main Inventory screen.
2. System displays list or dropdown of locations.
3. User clicks a location name.
4. System retrieves all food items stored in that location.
5. System updates the Items List to display those items.

Variation 1 – Location Has No Items

1.1 Items list displays empty table.

UC4 – Add Food Item

Actor: User

Description: The user adds a new food item using a GUI form.

Steps:

1. User clicks Add Item button.
2. GUI displays Add Item form containing:
    - Textfield: Name
    - Numeric field: Quantity
    - Dropdown: Location
    - Dropdown: Category
    - Date Picker: Expiration Date
    - Buttons: Save, Cancel
3. User enters item name and quantity.
4. User selects a location.
5. User optionally selects category.
6. User optionally selects expiration date.
7. User clicks Save.
8. System validates required fields.
9. System inserts new item into SQLite database.
10. System refreshes the items list.

Variation 1 – Missing Required Fields

1.1 User clicks Save with missing name, quantity, or location.

1.2 System displays popup: “Missing required fields.”

1.3 User clicks OK, popup closes.

Variation 2 – User Cancels

2.1 User clicks Cancel.

2.2 Form closes, no item created.

UC5 – Delete Food Item

Actor: User

Description: The user deletes a selected food item. No confirmation is required.

Steps:

1. User selects a location.
2. System displays the items list for that location.
3. User clicks an item to highlight it.
4. User clicks Delete Item button.
5. System deletes the item from the database.
6. System refreshes the item list.

Variation 1 – No Item Selected

1.1 User does not select any item.

1.2 User clicks Delete.

1.3 System shows popup: “No item selected.”

1.4 User clicks OK.

UC6 – Edit Food Item

Actor: User

Description: User edits an existing item using a GUI form.

Steps:

1. User selects a location to display items.
2. User selects an existing item.
3. User clicks Edit Item button.
4. System opens Edit Item form pre-filled with item data.
5. User changes one or more fields.
6. User clicks Save.
7. System validates input.
8. System updates the item in SQLite.
9. System refreshes the items list.

Variation 1 – No Item Selected

1.1 User does not select an item.

1.2 User clicks Edit Item.

1.3 System shows popup: “No item selected.”

Variation 2 – Cancel Edit

1.1 User clicks Cancel.

1.2 Form closes with no changes.

Variation 3 – Invalid Input

1.1 User enters invalid or empty name.

1.2 System displays popup: “Missing required fields.”

UC7 – Add Category

Actor: User

Description: User creates a new category.

Steps:

1. User clicks Categories button.
2. Category Management window opens.
3. User clicks Add Category.
4. GUI shows Add Category form (textfield + Save/Cancel).
5. User enters category name.
6. User clicks Save.
7. System validates name.
8. System inserts category into database.
9. System updates category list and dropdowns.

Variation 1 – Empty Name

1.1 Name field empty.

1.2 User clicks Save.

1.3 System shows error: “Missing required fields.”

Variation 2 – Duplicate Name

2.1 System detects duplicate.

2.3 System shows error: “Category name already exists.”

Variation 3 – Cancel

3.1 User clicks Cancel.

3.2 Form closes, nothing added.

UC8 – Delete Category

Actor: User

Description: User deletes a category; items using it are set to None.

Steps:

1. User opens Category Management window.
2. User selects a category.
3. User clicks Delete Category.
4. System shows confirmation popup:

   “Warning: All items will be set to “None” when removing category. Do you wish to continue?”
5. User clicks Yes.
6. System updates all items using that category to category=None.
7. System deletes category from database.
8. System refreshes category list.

Variation 1 – No Category Selected

1.1 User fails to select.

1.2 User clicks Delete.

1.3 Popup: “No category selected.”

Variation 2 – Cancel

2.1 User clicks No.

2.2 Popup closes, no deletion occurs.

UC9 – Edit Category

Actor: User

Description: User renames an existing category.

Steps:

1. User opens Category Management window.
2. User selects a category.
3. User clicks Edit Category.
4. GUI shows Edit Category form pre-filled.
5. User updates name.
6. User clicks Save.
7. System validates new name.
8. System updates database.
9. System refreshes the category list and dropdowns.

Variation 1 – No Category Selected

1.1 User clicks Edit without a selection.

1.2 System shows popup “No category selected”.

Variation 2 – Empty Name

2.1 New name empty.

2.2 System shows error popup “Missing required fields”.

Variation 3 – Duplicate Name

3.1 System detects duplicate.

3.2 System shows error: “Category name already exists.”

Variation 4 – Cancel

4.1 User clicks Cancel.

4.2 Edit form closes.

UC10 – Filter Items By Freshness

Actor: User

Description: The user filters displayed food items by enabling or disabling freshness rules using checkboxes. Each checkbox corresponds to a rule applied in a database query. No labels or classifications are assigned to the items; results are purely query-based.

Steps:

1. User selects a location so the items list is displayed.
2. System displays three checkboxes above the list:
   - Fresh (checked by default)
   - Expiring Soon (checked by default)
   - Expired (checked by default)
3. Because all three checkboxes are initially checked, the system runs a query that returns all items in the selected location.
4. User unchecks one of the checkboxes (e.g., Expired).
5. System determines the set of active rules based on checked boxes.
6. System constructs and runs a query that returns only items whose expiration dates satisfy at least one of the active rules.
7. System refreshes the items list to display only items matching the active filters.

Variation 1 – Multiple Filters Disabled

1.1 User unchecks Expired and Expiring Soon.

1.2 Remaining active rule is Fresh.

1.3 System runs query only for:

expiration_date > today + 30 days

1.4 Only fresh items are displayed.

Variation 2 – All Filters Disabled

2.1 User unchecks all three checkboxes.

2.2 There are no active rules.

2.3 System displays an empty items list.

Variation 3 – Filters Re-enabled

3.1 User re-checks one or more checkboxes.

3.2 Active rule set is updated.

3.3 System reruns the query based on the new set of rules.

3.4 Items list updates to reflect the newly applied filters.

Variation 4 – Items Without Expiration Date

4.1 Items list contains items with no expiration date.

4.2 Rule is made to include items with NULL values to the fresh query.

4.3 Items with no expiration date are listed when the fresh box is checked.

UC11 – Order Food Items

Actor: User

Description: The user changes the display order of food items using sorting arrows above the item list.

Steps:

1. User selects a location so the food items list is visible.
2. System displays the items list with column headers for:
   - Name
   - Category
   - Quantity

   Each header includes an up/down arrow for ordering.
3. User clicks the up arrow next to the Name column.
4. System sorts all visible food items alphabetically by name in ascending order (A–Z).
5. System refreshes the items list in the new order.

Variation 1 – Reverse Name Ordering

1.1 User clicks the down arrow next to the Name column.

1.2 System sorts items alphabetically in descending order (Z–A).

1.3 System refreshes the list.

Variation 2 – Order by Category

2.1 User clicks the up or down arrow next to the Category column.

2.2 System sorts items alphabetically by category name.

2.3 System refreshes the list.

Variation 3 – Order by Quantity

3.1 User clicks the up arrow next to the Quantity column.

3.2 System sorts items by quantity in ascending numeric order.

3.3 System refreshes the list.

3.4 User clicks the down arrow next to the Quantity column.

3.5 System sorts items by quantity in descending numeric order.

UC12 – Search for Food Item

Actor: User

Description: User searches by typing into a search bar and pressing search button.

Steps:

1. User views Items List.
2. User clicks inside Search Bar above the list.
3. User types a name or partial name.
4. User presses “Search” button.
5. System runs on-submit query.
6. System updates items list to show only matching items.

Variation 1 – User Clears Search

1.1 User presses “Clear” button.

1.2 System restores full list.

UC13 – Create Grocery List

Actor: User

Description: The user opens a temporary grocery list tool, types items, and optionally saves to a local PDF. The list is not stored in the application database.

Steps:

1. User clicks Grocery List button on the main screen.
2. System opens a Grocery List window containing:
   - A multi-line text area (initially empty)
   - A “Save” button
   - A “Print” button
   - A “Close” button
3. User types grocery items or notes into the text area.
4. User clicks Save.
5. System opens a Save As dialog (file chooser) configured to export as PDF.
6. User chooses a folder, enters a file name, and confirms Save.
7. System generates a PDF file with the contents of the text area in the chosen location.

Variation 1 – User Cancels Save Dialog

1.1 User clicks Cancel in the Save dialog.

1.2 No file is created; Grocery List window stays open.

Variation 2 – User Closes Grocery List Without Saving

2.1 User clicks Close button instead of Save or Print.

2.2 System shows confirmation popup: “Contents will be discarded. Do you wish to continue?”

2.3 User clicks yes

2.4 Grocery List window closes. (Stays open if no is clicked)

2.5 No data is stored in the application; contents are discarded.

UC14 – Print Grocery List

Actor: User

Description: The user prints the current contents of the grocery list text area directly, without storing it in the application.

Steps:

1. User has the Grocery List window open with text in the text area (from UC12).
2. User clicks Print button.
3. System opens the system Print dialog.
4. User selects printer and print options (pages, copies, etc.).
5. User clicks Print in the dialog.
6. System sends the grocery list contents to the selected printer.

Variation 1 – User Cancels Print Dialog

1.1 User clicks Cancel in the Print dialog.

1.2 No print job is sent and Grocery List window remains open.

3. Glossary (UPDATED)

- **Food Item:** A grocery product stored in the system inventory.

- **Category:** A label used to classify food items (e.g., Dairy, Snacks).
- **Location:** A physical storage space where food items are kept (e.g., Pantry, Fridge, Freezer).
- **Inventory:** The collection of all food items currently stored in the system.
- **Freshness Filter:** A rule-based filter that displays items based on expiration date conditions.
- **Expiration Date:** A calendar date indicating when a food item is no longer considered usable.
- **Grocery List:** A temporary text-based tool used to manually create a shopping list that can be saved to PDF or printed.
- **Database:** The structured storage system (SQLite) used to store food items, categories, and locations.
- **SQLite:** A lightweight, file-based relational database system used by the application.
- **Query:** A database command used to retrieve, filter, sort, insert, update, or delete data.
- **NULL:** A database value that represents the absence of data, used for items with no expiration date.
- **MVC (Model-View-Controller):** An architectural pattern that separates application data (Model), user interface (View), and control logic (Controller).
- **GUI (Graphical User Interface):** The visual interface through which the user interacts with the application.

- **PDF Export:** The process of saving the grocery list to a Portable Document Format (PDF) file on the local system.
- **Print Dialog:** The system-provided interface that allows the user to select a printer and print settings before printing.

4. UML Class Diagrams

4.1 Class Diagram: Core Domain Model

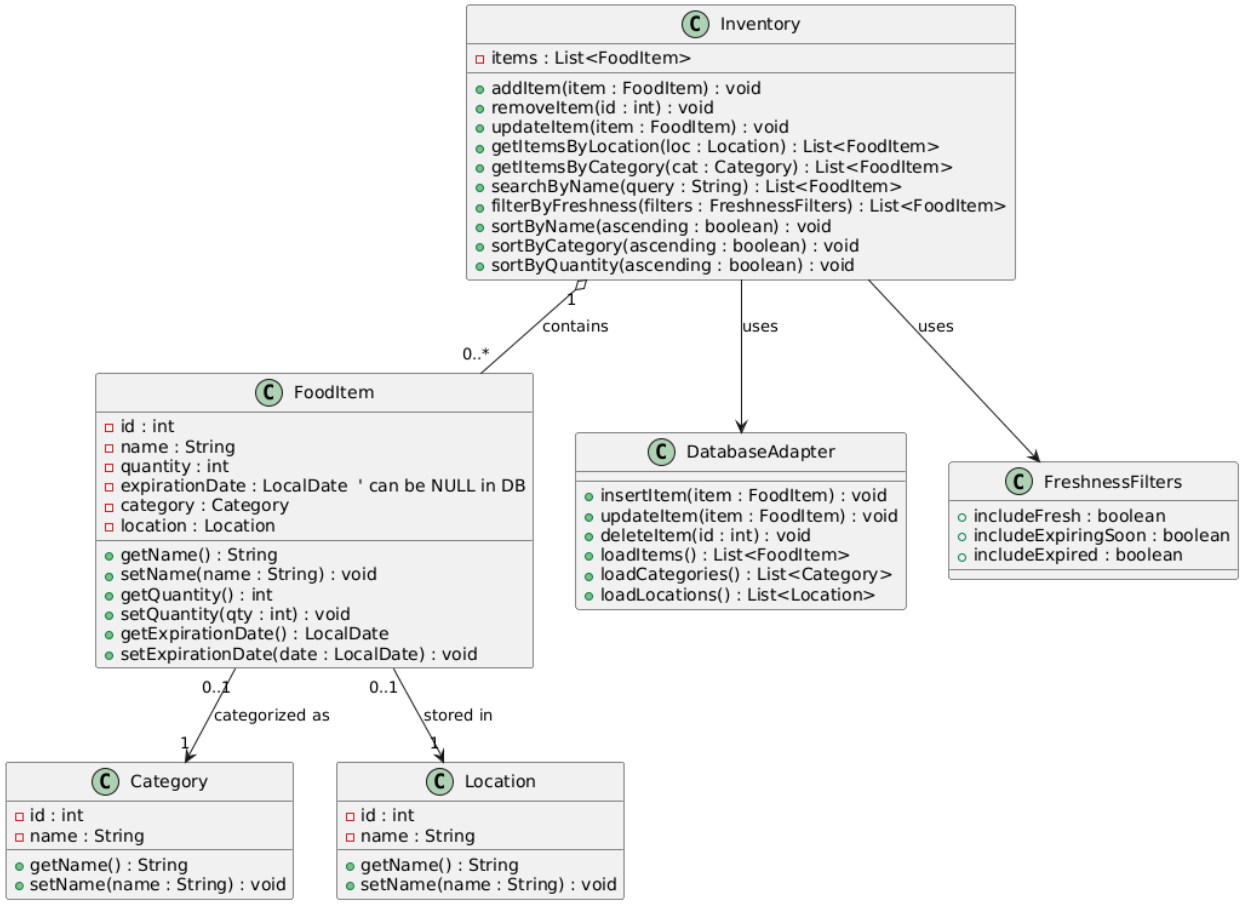
Overview: This class diagram represents the core data structures of the Grocery Inventory Manager. It models the main business objects used throughout the system, including FoodItem, Category, Location, Inventory, and DatabaseAdapter. These classes define how food items are stored, categorized, assigned to locations, and persisted in the SQLite database. This diagram focuses purely on the data layer and its relationships.

4.2 Class Diagram: Model View Controller

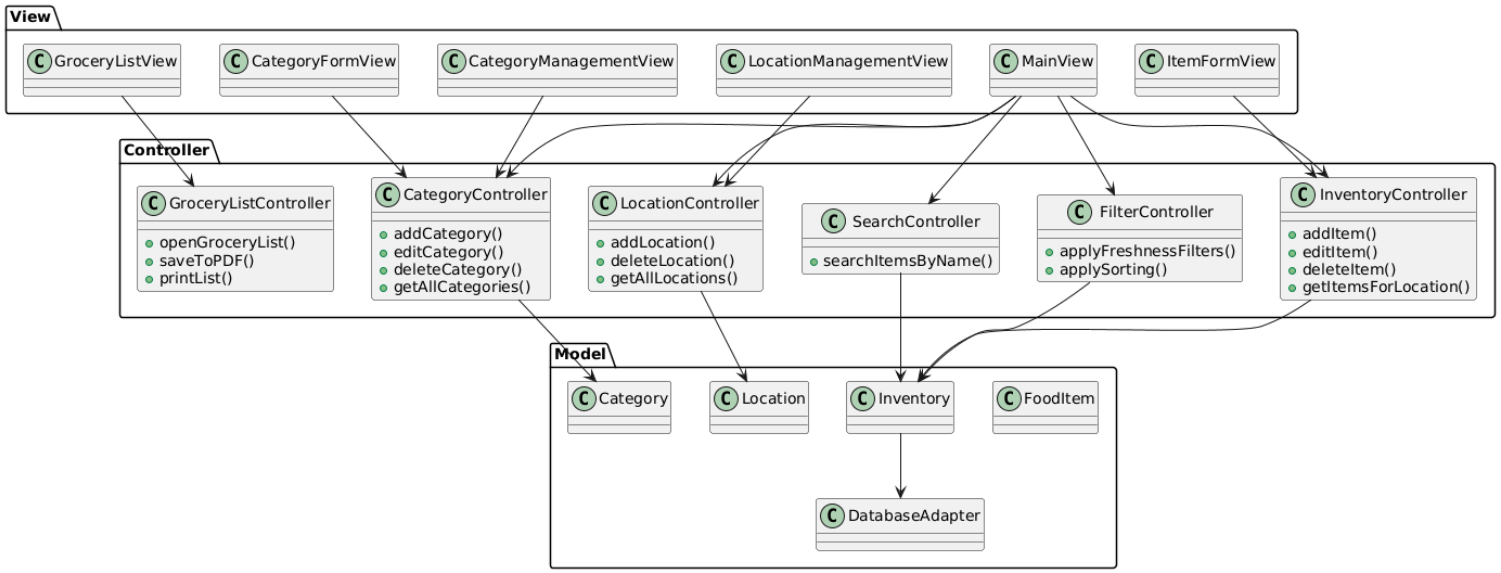
Overview: This class diagram illustrates the overall Model-View-Controller (MVC) architecture of the application. It shows how the system is divided into three layers:

- The Model layer contains the core data and database logic.
- The View layer contains all GUI screens and forms.

CD1 - Core Domain Model



CD2 - MVC Overview



- The Controller layer contains the application logic that connects user actions to system behavior.

This diagram demonstrates how Views communicate with Controllers and how Controllers interact with the Model.

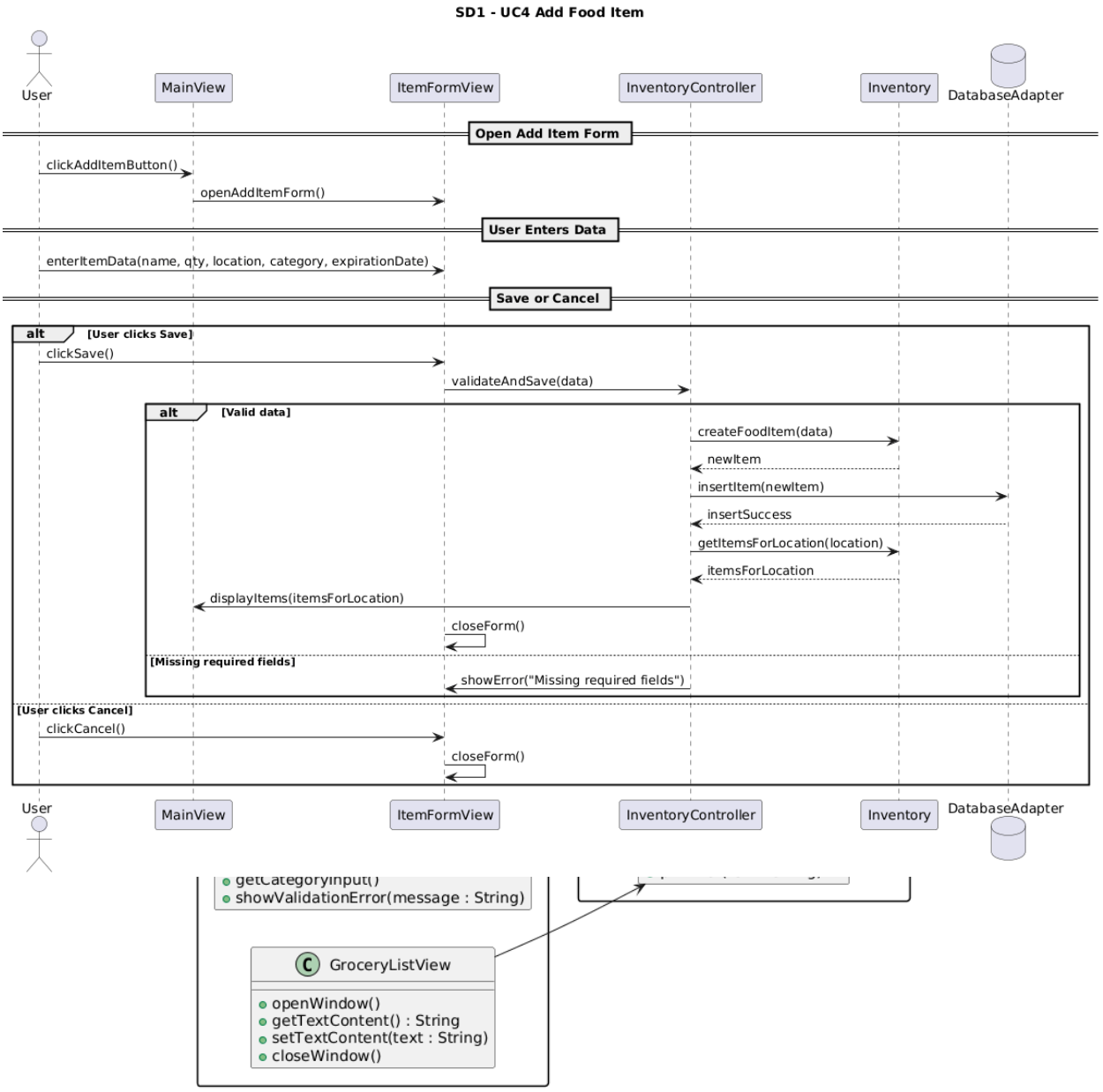
4.3 Class Diagram: GUI Management

Overview: This class diagram focuses specifically on the graphical user interface and its controllers. It shows the relationships between the main screens (such as MainView, LocationManagementView, CategoryManagementView, and GroceryListView) and their corresponding controllers. This diagram highlights how user interactions in the GUI are handled and routed to the appropriate controller logic.

5. UML Sequence Diagrams

5.1 Sequence Diagram: UC4 Add Food Item

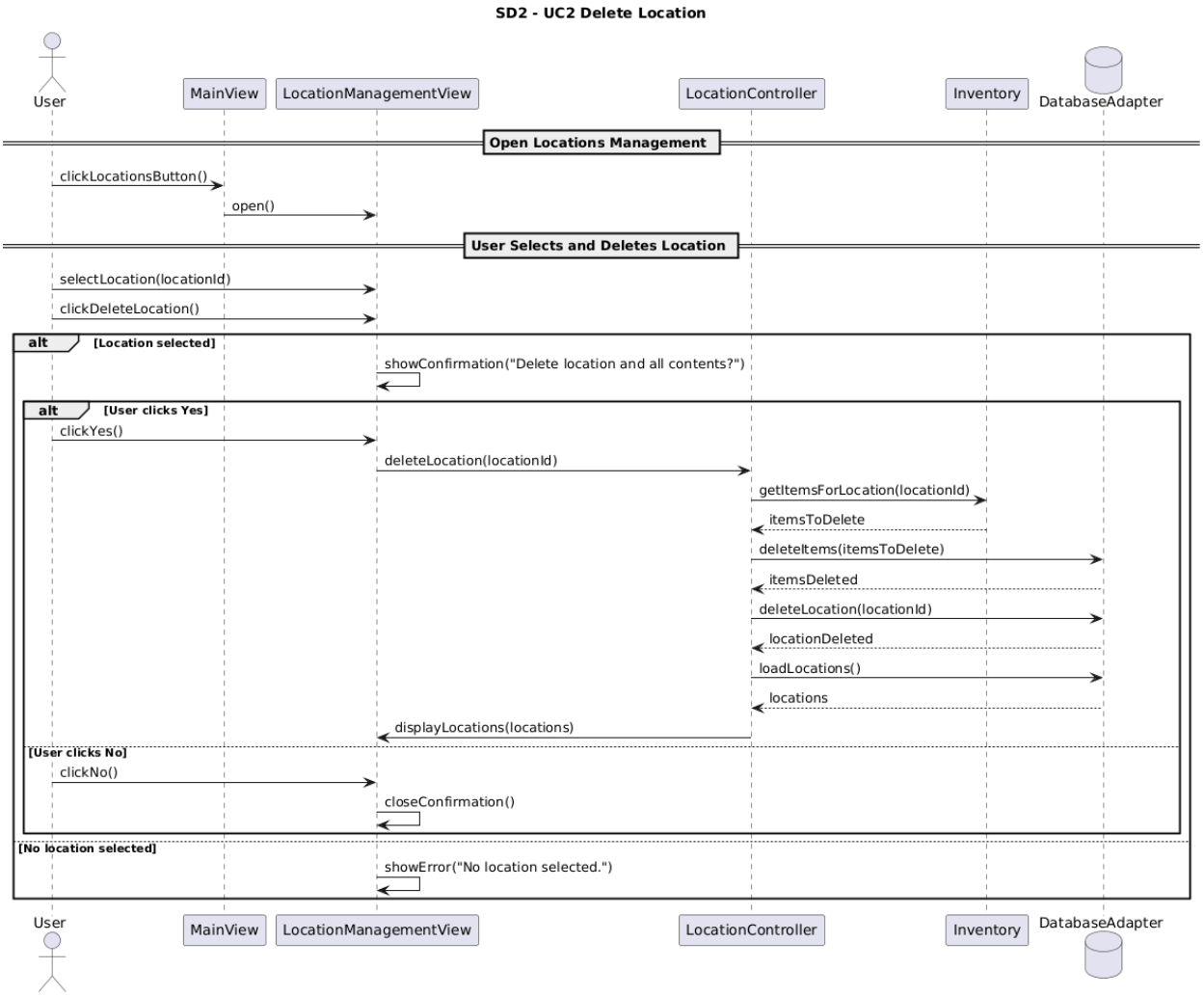
Overview: This sequence diagram shows how a new food item is added to the system. It tracks the interaction between the User, the item form, the controller, the inventory, and the database. The process includes opening the form, entering data, validating input, saving the item, and refreshing the item list. It also includes alternate flows for missing required fields and user cancellation.



5.2 Sequence Diagram: UC2 Delete Location

Overview: This sequence diagram shows how a location is deleted from the system. It covers the flow where the user opens the location management window, selects a location, clicks delete, confirms the deletion, and the system removes the location and

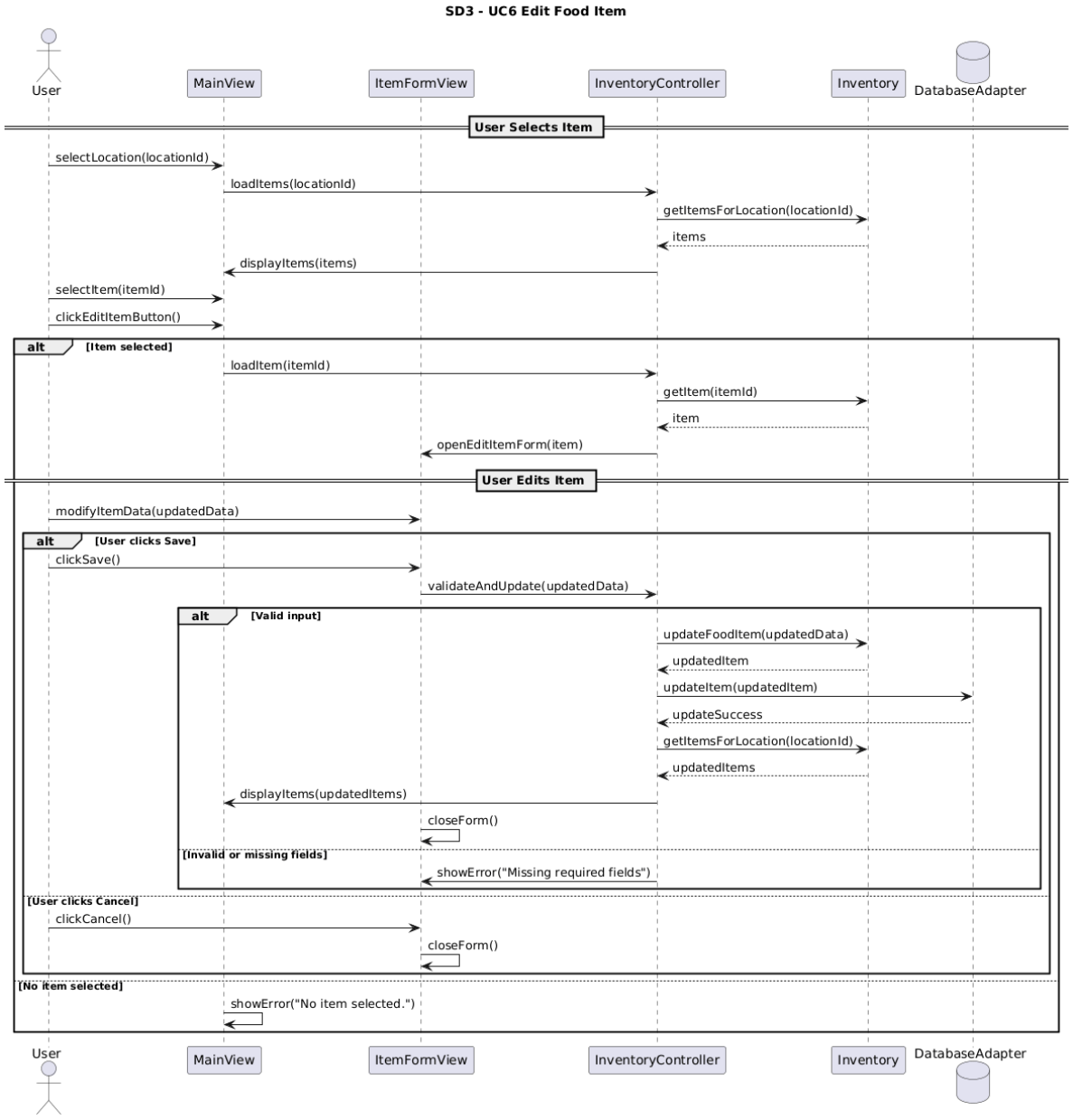
its items from the database. It also includes variations where the user cancels the confirmation dialog or clicks delete without selecting a location.



5.3 Sequence Diagram: UC6 Edit Food Item

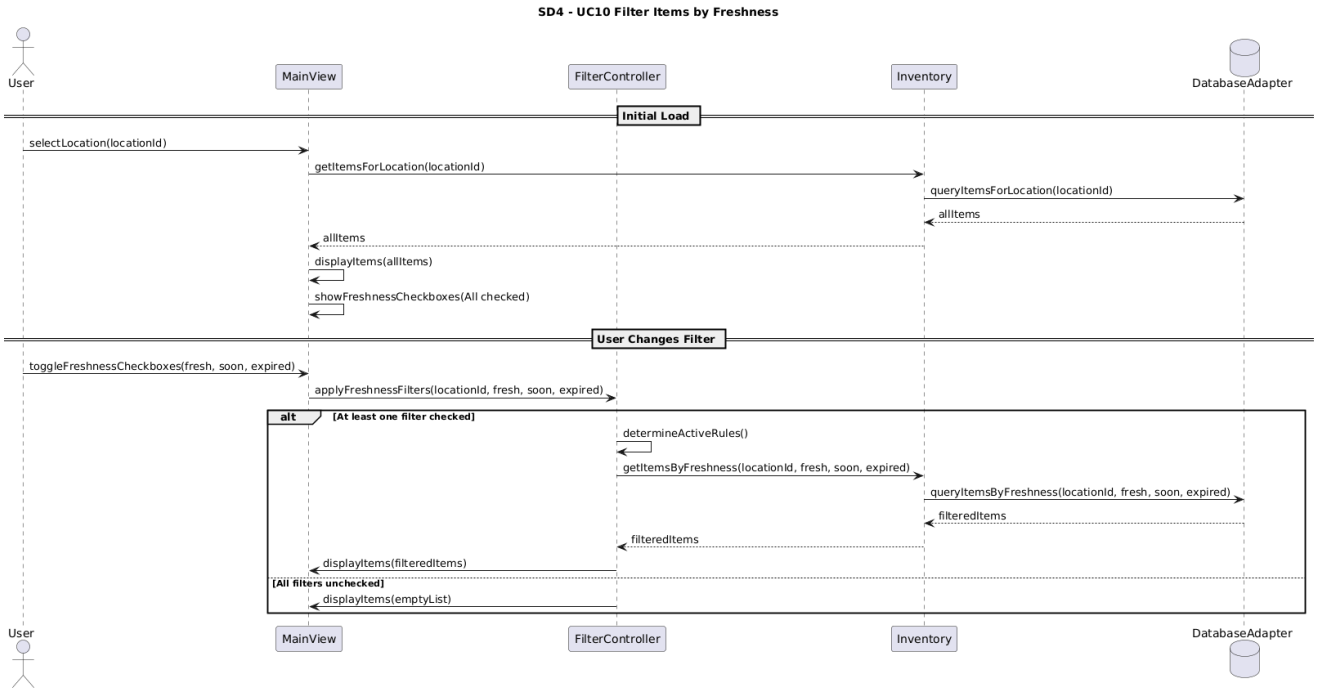
This sequence diagram shows how an existing food item is edited. It covers the user selecting an item, opening the edit form, updating the fields, and saving the changes. The controller validates the input, updates the item in the database, and then refreshes

the items list. It also includes alternate flows for no item selected, invalid input, and user cancellation.



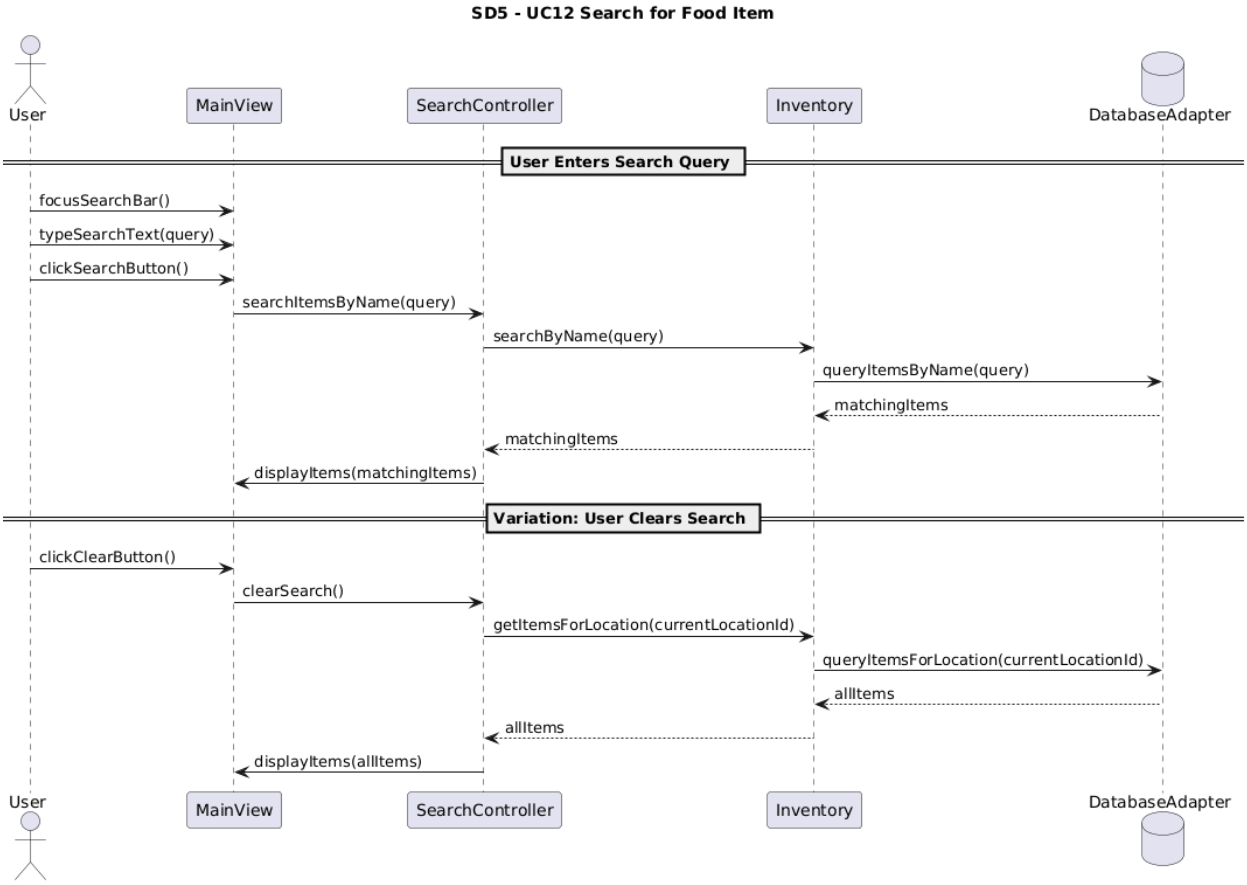
5.4 Sequence Diagram: UC10 Filter Items By Freshness

Overview: This sequence diagram shows how the system filters items by freshness when the user interacts with the freshness checkboxes. The user changes which checkboxes are active. The view notifies the filter controller, which determines the active rules, builds the corresponding query, and asks the database for matching items. The filtered items are then returned and displayed. A variation is included for the case where all filters are turned off and no items are shown.



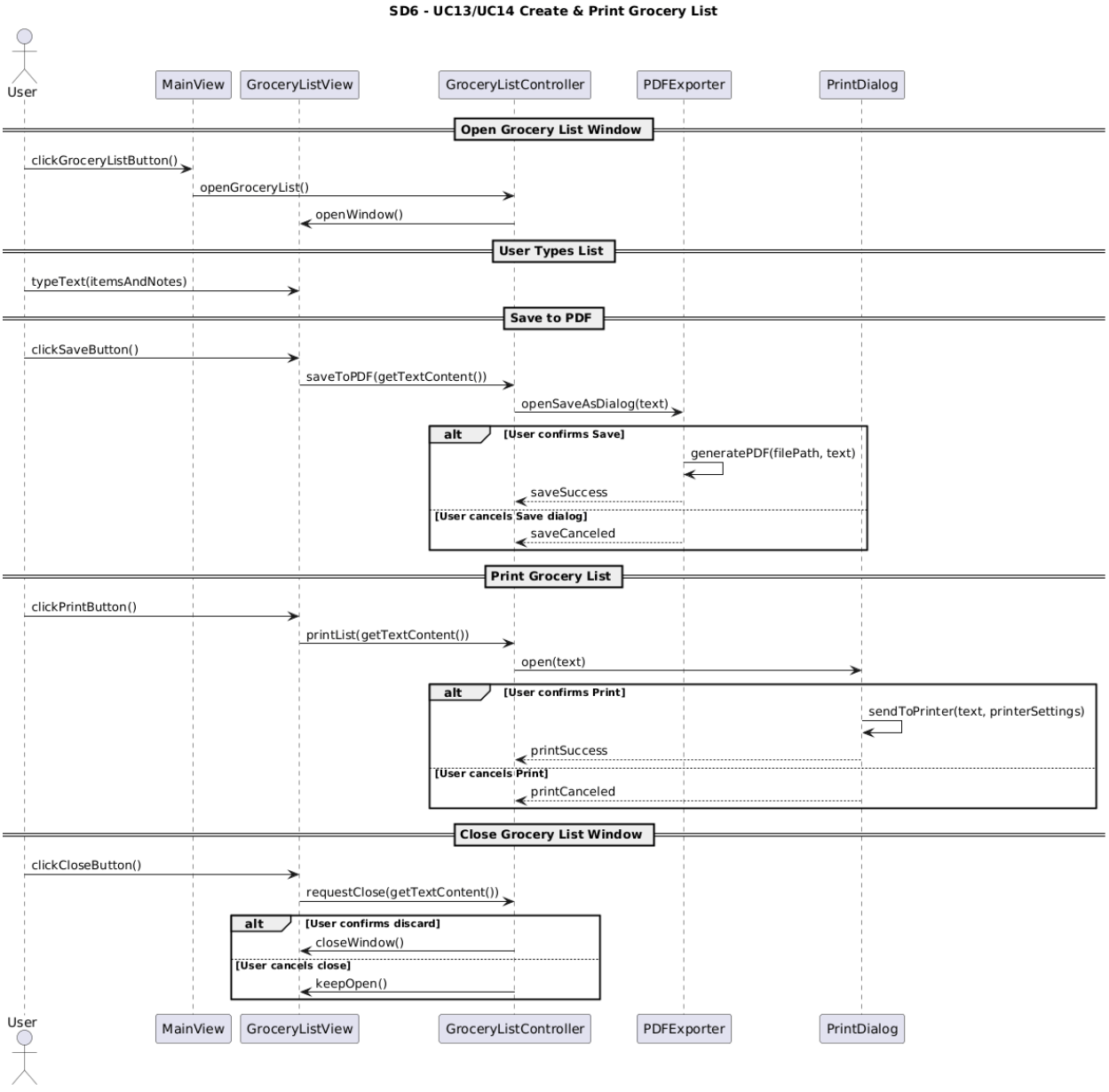
5.5 Sequence Diagram: UC12 Search For Food Item

Overview: This sequence diagram shows how the user searches for items by name. The user types text into the search bar and clicks the Search button. The view sends the query to the search controller, which asks the inventory and database for matching items. The results are then returned and displayed. A variation covers the Clear button, which resets the search and shows the full list again.



5.6 Sequence Diagram: UC13/14

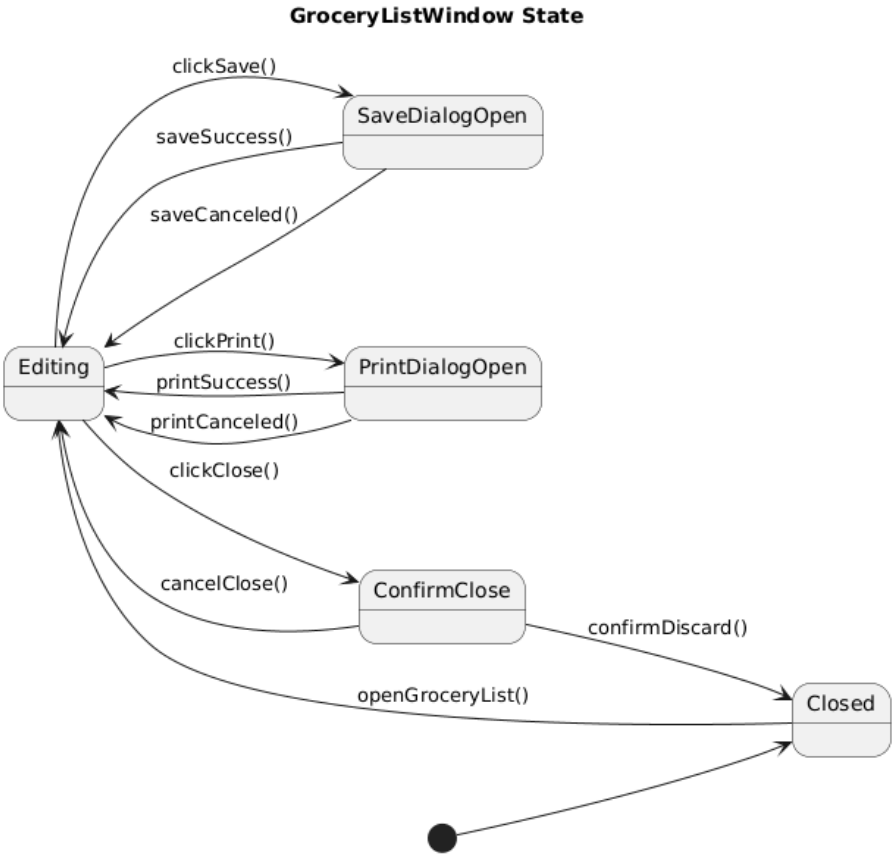
Overview: This sequence diagram shows how the grocery list tool is used. The user opens the grocery list window, types text, and can then either save the contents to a local PDF or print them. The view forwards user actions to the grocery list controller, which calls a PDF exporter for saving and the system print dialog for printing. Variations include canceling the Save As dialog or the Print dialog, and closing the window without saving.



6. UML State Diagrams

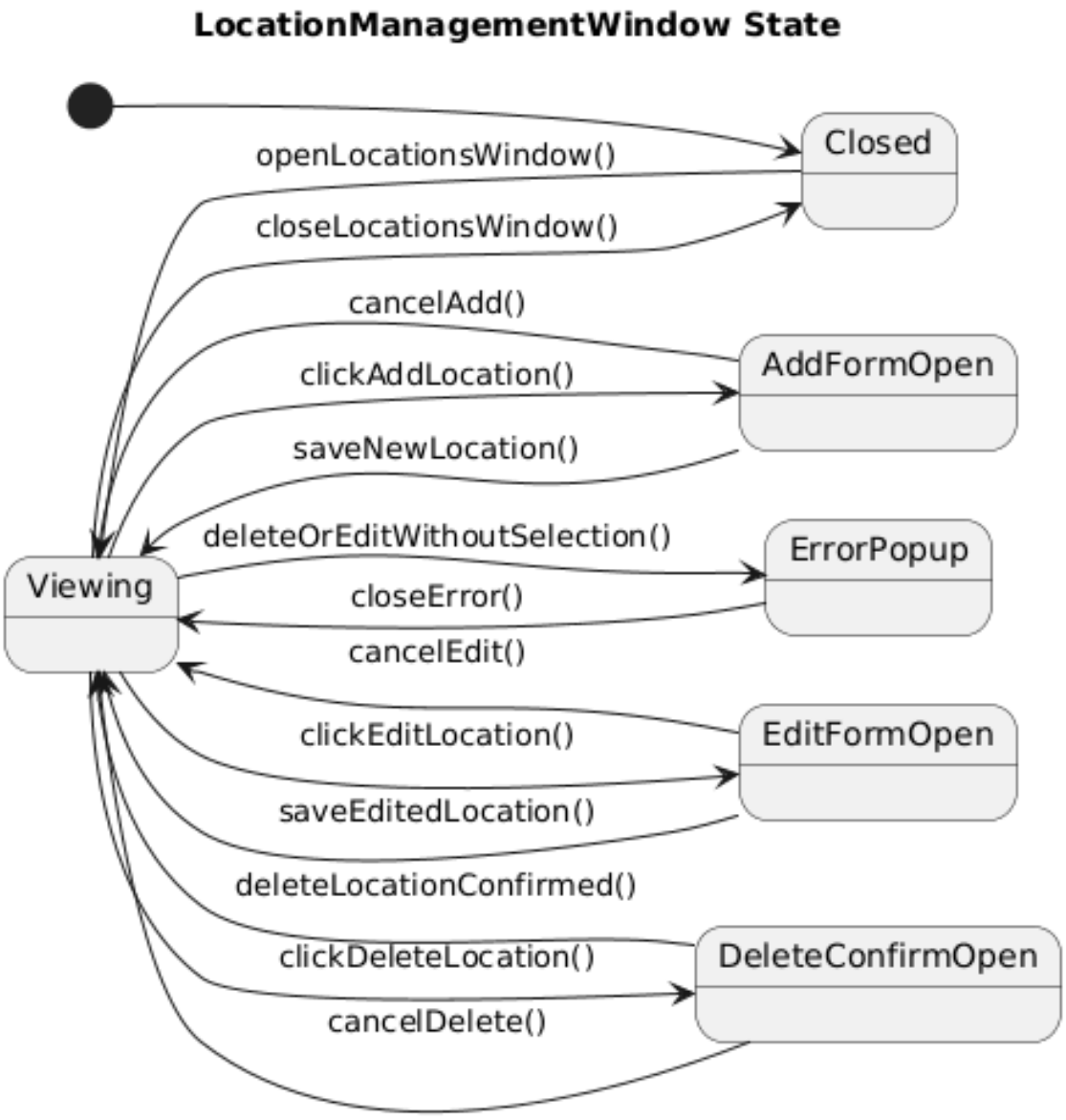
6.1 State Diagram: GroceryListWindow State

Overview: This state diagram shows how the grocery list window behaves over time. It starts closed, can be opened for editing, and then moves into temporary states when the user chooses to save to PDF, print, or close the window. The diagram focuses on the transitions between editing, save dialog, print dialog, and the confirmation prompt when closing without saving.



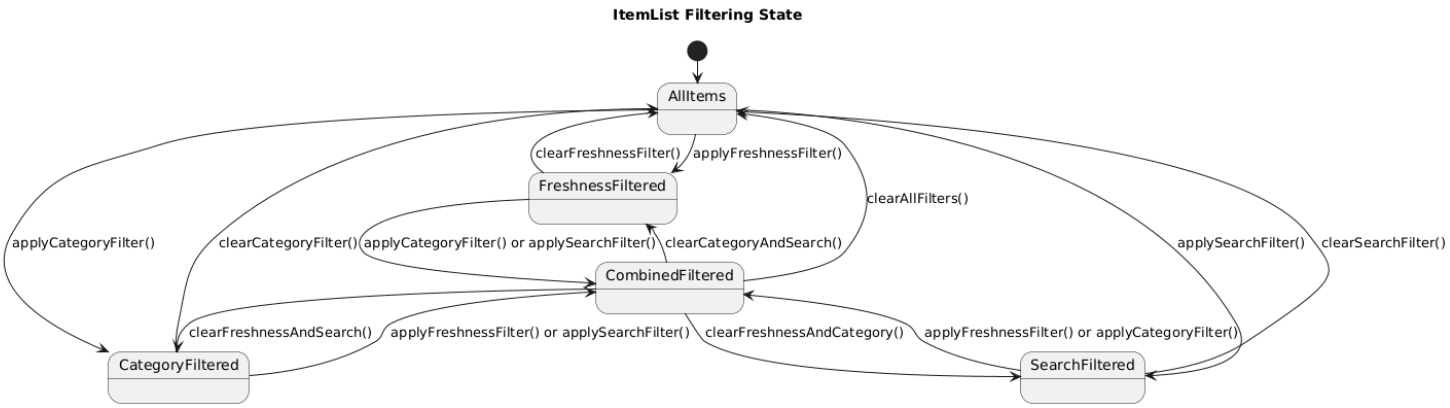
6.2 State Diagram: LocationManagementWindow State

Overview: This state diagram shows how the Location Management window behaves. It starts closed, can be opened from the main view, and then moves between viewing, add form, edit form, delete confirmation, and error popup states. It covers actions such as adding a location, deleting a location with confirmation, and showing errors when no location is selected.



6.3 State Diagram: ItemList Filtering State

Overview: This state diagram shows how the item list view behaves as the user applies and clears filters. The list can be in a default state showing all items, filtered by freshness, filtered by category, filtered by search text, or in a combined state when more than one filter is active. Transitions occur when the user changes checkboxes, category selection, search text, or clears filters.



7. Design Summary

The Grocery Inventory Manager is designed using the Model View Controller pattern to keep the system organized and easy to maintain. The Model contains the core data such as food items, categories, locations, and the inventory that manages them. The View includes all of the screens the user interacts with, such as the main item list, category and location management screens, item forms, and the grocery list tool. The Controller layer handles all the logic such as adding items, editing data, filtering, searching, and communicating with the database.

Three main class diagrams were created. One focuses on the core data of the system, another shows how the MVC structure is organized, and the third focuses on how the user interface connects to the controllers. Several sequence diagrams were made to show how key user actions work step by step, including adding items, deleting locations, editing items, filtering by freshness, searching for items, and using the grocery list tool. State diagrams were also created to show how different windows and views change over time, including the grocery list window, the location management window, and the item list when filters are applied.

Overall, the design shows how user actions move through the system, how data is stored and updated, and how the interface responds. The goal of the design is to keep the system easy to understand, easy to extend, and consistent in how it handles user input and data updates.

8. Testing

8.1 Objectives

The goal of the testing activities was to verify the correctness of the core business logic that drives the Grocery Inventory Manager application. Since the application separates logic from UI (using the MVC pattern), the tests focus on validating the model-layer behavior, such as:

- Freshness classification rules based on expiration dates
- Sorting behavior for items by name, category, and quantity
- Basic database initialization behavior

These automated tests also serve as regression checks, ensuring that future modifications (UI updates, database changes, refactoring) do not unintentionally break underlying logic that the GUI depends on.

8.2 Test Scope

Automated Unit Tests (JUnit 5)

Classes under test:

- model.Inventory
- model.FoodItem
- dao.DatabaseAdapter (lightweight initialization test)

Methods and behaviors validated:

Freshness Filtering

- Inventory.filterByFreshness(includeFresh, includeExpiringSoon, includeExpired)
- Items expiring more than 30 days from today are classified as fresh
- Items within 30 days are classified as expiring soon
- Items with past dates are expired
- Items with no expiration date (NULL) are treated as fresh

Test class:

InventoryFreshnessFilterTest

Sorting Logic

- Inventory.sortByName(...)
- Inventory.sortByCategory(...)
- Inventory.sortByQuantity(...)

These tests confirm that:

- Sorting works in ascending and descending orders
- Sorting uses the correct keys (category name, numeric quantity, etc.)
- Sorting results match expected lists independent of original order

Test class:

InventorySortingTest

Database Initialization

DatabaseAdapter.initializeDatabase()

This test confirms:

- A SQLite database file is created successfully

- Initialization does not throw errors
- The DAO is structurally correct for interacting with SQLite

Test class:

DatabaseAdapterTest

Out of Scope (Not Automated)

The following areas were tested manually due to their graphical or interactive nature:

- Swing GUI behaviors (MainView, ItemFormView, LocationManagementView, CategoryManagementView)
- Table sorting through the Swing TableRowSorter integration
- Search and filter interaction together in the main table
- Grocery List window's Save and Print dialogs
- SQLite CRUD operations triggered from controller actions

These were validated through exploratory, click-through testing inside the running application.

6.3 Test Environment

- IDE: NetBeans
- JDK: 24 (Default Platform)
- Testing Framework: JUnit 5 (JUnit Jupiter)
- Build Tool: Maven

Main.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 */

package com.james.groceryinventorymanager;
import com.james.groceryinventorymanager.dao.DatabaseAdapter;
import com.james.groceryinventorymanager.model.Inventory;
import
com.james.groceryinventorymanager.controller.InventoryController
;
import com.james.groceryinventorymanager.view.MainView;

import javax.swing.SwingUtilities;

/**
 *
 * @author james
 */

public class Main {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            String dbFilePath = "grocery.db";

            DatabaseAdapter dbAdapter = new
DatabaseAdapter(dbFilePath);
            dbAdapter.initializeDatabase();

            Inventory inventory = new Inventory();
            // controller will load everything
            InventoryController inventoryController = new
InventoryController(inventory, dbAdapter);

            MainView mainView = new
MainView(inventoryController);
            mainView.setVisible(true);
        });
    }
}
```

Category.java

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.model;
import java.util.Objects;

/**
 *
 * @author james
 */

public class Category {

    private int id;
    private String name;

    public Category(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Category(String name) {
        this(0, name);
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Category)) return false;
        Category category = (Category) o;
        if (this.id != 0 && category.id != 0) {
            return this.id == category.id;
        }
        return Objects.equals(name, category.name);
    }

    @Override
    public int hashCode() {
        if (id != 0) {
            return Integer.hashCode(id);
        }
        return Objects.hash(name);
    }

    @Override
    public String toString() {
        return name;
    }
}
```

FoodItem.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.model;
import java.time.LocalDate;
import java.util.Objects;

/**
 *
 * @author james
 */

public class FoodItem {
```

```
    private int id; // database id, 0 if not yet persisted
    private String name;
    private int quantity;
    private LocalDate expirationDate; // can be null
    private Category category;        // can be null for "None"
    private Location location;        // should not be null

    public FoodItem(int id,
                    String name,
                    int quantity,
                    LocalDate expirationDate,
                    Category category,
                    Location location) {
        this.id = id;
        this.name = name;
        this.quantity = quantity;
        this.expirationDate = expirationDate;
        this.category = category;
        this.location = location;
    }

    public FoodItem(String name,
                    int quantity,
                    LocalDate expirationDate,
                    Category category,
                    Location location) {
        this(0, name, quantity, expirationDate, category,
location);
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```java
    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public LocalDate getExpirationDate() {
        return expirationDate;
    }

    public void setExpirationDate(LocalDate expirationDate) {
        this.expirationDate = expirationDate;
    }

    public Category getCategory() {
        return category;
    }

    public void setCategory(Category category) {
        this.category = category;
    }

    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof FoodItem)) return false;
        FoodItem foodItem = (FoodItem) o;
        // if id is set, use id for equality
        if (this.id != 0 && foodItem.id != 0) {
            return this.id == foodItem.id;
        }
        // fallback for non persisted items
        return Objects.equals(name, foodItem.name)
                && quantity == foodItem.quantity
                && Objects.equals(expirationDate,
foodItem.expirationDate)
```

```
                && Objects.equals(category, foodItem.category)
                && Objects.equals(location, foodItem.location);
    }

    @Override
    public int hashCode() {
        if (id != 0) {
            return Objects.hash(id);
        }
        return Objects.hash(name, quantity, expirationDate,
category, location);
    }

    @Override
    public String toString() {
        return "FoodItem{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", quantity=" + quantity +
                ", expirationDate=" + expirationDate +
                ", category=" + (category != null ?
category.getName() : "None") +
                ", location=" + (location != null ?
location.getName() : "None") +
                '}';
    }
}
```

Inventory.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.model;
import com.james.groceryinventorymanager.model.Location;
import com.james.groceryinventorymanager.model.Category;
import com.james.groceryinventorymanager.model.FoodItem;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
```

```java
import java.util.List;
import java.util.stream.Collectors;

/**
 *
 * @author james
 */

public class Inventory {

    private final List<FoodItem> items = new ArrayList<>();

    public void clear() {
        items.clear();
    }

    public void addItem(FoodItem item) {
        items.add(item);
    }

    public void removeItem(FoodItem item) {
        items.remove(item);
    }

    public void removeItemById(int id) {
        items.removeIf(i -> i.getId() == id);
    }

    public List<FoodItem> getAllItems() {
        return new ArrayList<>(items);
    }

    public List<FoodItem> getItemsByLocation(Location location)
{
        return items.stream()
                .filter(i -> i.getLocation() != null &&
i.getLocation().equals(location))
                .collect(Collectors.toList());
    }

    public List<FoodItem> getItemsByCategory(Category category)
{
        return items.stream()
                .filter(i -> i.getCategory() != null &&
i.getCategory().equals(category))
                .collect(Collectors.toList());
```

```
    }

    public List<FoodItem> searchByName(String query) {
        String lower = query.toLowerCase();
        return items.stream()
                .filter(i -> i.getName() != null &&
i.getName().toLowerCase().contains(lower))
                .collect(Collectors.toList());
    }

    /**
     * Filter items by freshness rules.
     *
     * includeFresh: expiration_date is null or > today + 30
days
     * includeExpiringSoon: expiration_date > today and <= today
+ 30 days
     * includeExpired: expiration_date <= today
     */
    public List<FoodItem> filterByFreshness(boolean
includeFresh,
                                            boolean
includeExpiringSoon,
                                            boolean
includeExpired) {
        LocalDate today = LocalDate.now();
        LocalDate in30Days = today.plusDays(30);

        return items.stream()
                .filter(item -> {
                    LocalDate exp = item.getExpirationDate();

                    // Treat null expiration as Fresh
                    if (exp == null) {
                        return includeFresh;
                    }

                    boolean isExpired = !exp.isAfter(today); //
exp <= today
                    boolean isExpiringSoon = exp.isAfter(today)
&& !exp.isAfter(in30Days);
                    boolean isFreshItem = exp.isAfter(in30Days);

                    return (includeFresh && isFreshItem)
                            || (includeExpiringSoon &&
isExpiringSoon)
```

```
                            || (includeExpired && isExpired);
                })
                .collect(Collectors.toList());
    }

    public List<FoodItem> sortByName(List<FoodItem> source,
boolean ascending) {
        List<FoodItem> copy = new ArrayList<>(source);
        Comparator<FoodItem> cmp =
Comparator.comparing(FoodItem::getName,
                String.CASE_INSENSITIVE_ORDER);
        if (!ascending) {
            cmp = cmp.reversed();
        }
        copy.sort(cmp);
        return copy;
    }

    public List<FoodItem> sortByCategory(List<FoodItem> source,
boolean ascending) {
        List<FoodItem> copy = new ArrayList<>(source);
        Comparator<FoodItem> cmp = Comparator.comparing(
                item -> item.getCategory() != null ?
item.getCategory().getName() : "",
                String.CASE_INSENSITIVE_ORDER
        );
        if (!ascending) {
            cmp = cmp.reversed();
        }
        copy.sort(cmp);
        return copy;
    }

    public List<FoodItem> sortByQuantity(List<FoodItem> source,
boolean ascending) {
        List<FoodItem> copy = new ArrayList<>(source);
        Comparator<FoodItem> cmp =
Comparator.comparingInt(FoodItem::getQuantity);
        if (!ascending) {
            cmp = cmp.reversed();
        }
        copy.sort(cmp);
        return copy;
    }
}
```

Location.java

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.model;
import java.util.Objects;
/**
 *
 * @author james
 */

public class Location {

    private int id;
    private String name;

    public Location(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Location(String name) {
        this(0, name);
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Location)) return false;
        Location that = (Location) o;
        if (this.id != 0 && that.id != 0) {
            return this.id == that.id;
        }
        return Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        if (id != 0) {
            return Integer.hashCode(id);
        }
        return Objects.hash(name);
    }

    @Override
    public String toString() {
        return name;
    }
}
```

DatabaseAdapter.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.dao;
import com.james.groceryinventorymanager.model.Category;
import com.james.groceryinventorymanager.model.FoodItem;
import com.james.groceryinventorymanager.model.Inventory;
import com.james.groceryinventorymanager.model.Location;

import java.sql.*;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
/**
 *
```

```
 * @author james
 */

public class DatabaseAdapter {

    private final String url;

    public DatabaseAdapter(String dbFilePath) {
        // Example: "jdbc:sqlite:grocery.db"
        this.url = "jdbc:sqlite:" + dbFilePath;
    }

    private Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url);
    }

    /**
     * Create tables if they do not exist.
     */
    public void initializeDatabase() {
        String createLocations = """
                CREATE TABLE IF NOT EXISTS locations (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    name TEXT NOT NULL UNIQUE
                );
                """;

        String createCategories = """
                CREATE TABLE IF NOT EXISTS categories (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    name TEXT NOT NULL UNIQUE
                );
                """;

        String createItems = """
                CREATE TABLE IF NOT EXISTS items (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    name TEXT NOT NULL,
                    quantity INTEGER NOT NULL,
                    expiration_date TEXT NULL,
                    category_id INTEGER NULL,
                    location_id INTEGER NOT NULL,
                    FOREIGN KEY (category_id) REFERENCES
categories(id) ON DELETE SET NULL,
                    FOREIGN KEY (location_id) REFERENCES
locations(id) ON DELETE CASCADE
```

```
                );
                """;

        try (Connection conn = getConnection();
             Statement stmt = conn.createStatement()) {

            stmt.execute(createLocations);
            stmt.execute(createCategories);
            stmt.execute(createItems);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // ==============================
    // Locations
    // ==============================

    public List<Location> loadLocations() {
        List<Location> locations = new ArrayList<>();
        String sql = "SELECT id, name FROM locations ORDER BY
name ASC";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql);
             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                locations.add(new Location(id, name));
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }

        return locations;
    }

    public Location insertLocation(Location location) throws
SQLException {
        String sql = "INSERT INTO locations(name) VALUES (?)";

        try (Connection conn = getConnection();
```

```
             PreparedStatement ps = conn.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS)) {

            ps.setString(1, location.getName());
            ps.executeUpdate();

            try (ResultSet keys = ps.getGeneratedKeys()) {
                if (keys.next()) {
                    int id = keys.getInt(1);
                    location.setId(id);
                }
            }
        }

        return location;
    }

    public void deleteLocation(int locationId) throws
SQLException {
        String sql = "DELETE FROM locations WHERE id = ?";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql))
{

            ps.setInt(1, locationId);
            ps.executeUpdate();
        }
    }

    // ==============================
    // Categories
    // ==============================

    public List<Category> loadCategories() {
        List<Category> categories = new ArrayList<>();
        String sql = "SELECT id, name FROM categories ORDER BY
name ASC";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql);
             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
```

```
                categories.add(new Category(id, name));
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }

        return categories;
    }

    public Category insertCategory(Category category) throws
SQLException {
        String sql = "INSERT INTO categories(name) VALUES (?)";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS)) {

            ps.setString(1, category.getName());
            ps.executeUpdate();

            try (ResultSet keys = ps.getGeneratedKeys()) {
                if (keys.next()) {
                    int id = keys.getInt(1);
                    category.setId(id);
                }
            }
        }

        return category;
    }

    public void updateCategory(Category category) throws
SQLException {
        String sql = "UPDATE categories SET name = ? WHERE id
= ?";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql))
{

            ps.setString(1, category.getName());
            ps.setInt(2, category.getId());
            ps.executeUpdate();
        }
    }
```

```
    public void deleteCategory(int categoryId) throws
SQLException {
        String sql = "DELETE FROM categories WHERE id = ?";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql))
{

            ps.setInt(1, categoryId);
            ps.executeUpdate();
        }
    }

    // ================================
    // Items
    // ================================

    /**
     * Load all items for a given location.
     */
    public List<FoodItem> loadItemsForLocation(Location
location,
                                               List<Category>
knownCategories,
                                               List<Location>
knownLocations) {
        List<FoodItem> items = new ArrayList<>();

        String sql = """
                SELECT i.id, i.name, i.quantity,
i.expiration_date,
                       i.category_id, i.location_id
                FROM items i
                WHERE i.location_id = ?
                ORDER BY i.name ASC
                """;

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql))
{

            ps.setInt(1, location.getId());

            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
```

```
                    int id = rs.getInt("id");
                    String name = rs.getString("name");
                    int quantity = rs.getInt("quantity");
                    String expText =
rs.getString("expiration_date");
                    Integer categoryId =
rs.getObject("category_id") != null
                            ? rs.getInt("category_id")
                            : null;
                    int locationId = rs.getInt("location_id");

                    LocalDate expDate = null;
                    if (expText != null) {
                        expDate = LocalDate.parse(expText); //
expects ISO yyyy-MM-dd
                    }

                    Category category = null;
                    if (categoryId != null) {
                        for (Category c : knownCategories) {
                            if (c.getId() == categoryId) {
                                category = c;
                                break;
                            }
                        }
                    }

                    Location loc = null;
                    for (Location l : knownLocations) {
                        if (l.getId() == locationId) {
                            loc = l;
                            break;
                        }
                    }

                    FoodItem item = new FoodItem(id, name,
quantity, expDate, category, loc);
                    items.add(item);
                }
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }

        return items;
```

```
    }

    public FoodItem insertItem(FoodItem item) throws
SQLException {
        String sql = """
                INSERT INTO items(name, quantity,
expiration_date, category_id, location_id)
                VALUES (?, ?, ?, ?, ?)
                """;

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS)) {

            ps.setString(1, item.getName());
            ps.setInt(2, item.getQuantity());

            if (item.getExpirationDate() != null) {
                ps.setString(3,
item.getExpirationDate().toString());
            } else {
                ps.setNull(3, Types.VARCHAR);
            }

            if (item.getCategory() != null &&
item.getCategory().getId() != 0) {
                ps.setInt(4, item.getCategory().getId());
            } else {
                ps.setNull(4, Types.INTEGER);
            }

            ps.setInt(5, item.getLocation().getId());

            ps.executeUpdate();

            try (ResultSet keys = ps.getGeneratedKeys()) {
                if (keys.next()) {
                    int id = keys.getInt(1);
                    item.setId(id);
                }
            }
        }

        return item;
    }
```

```
    public void updateItem(FoodItem item) throws SQLException {
        String sql = """
                UPDATE items
                SET name = ?, quantity = ?, expiration_date = ?,
category_id = ?, location_id = ?
                WHERE id = ?
                """;

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql))
{

            ps.setString(1, item.getName());
            ps.setInt(2, item.getQuantity());

            if (item.getExpirationDate() != null) {
                ps.setString(3,
item.getExpirationDate().toString());
            } else {
                ps.setNull(3, Types.VARCHAR);
            }

            if (item.getCategory() != null &&
item.getCategory().getId() != 0) {
                ps.setInt(4, item.getCategory().getId());
            } else {
                ps.setNull(4, Types.INTEGER);
            }

            ps.setInt(5, item.getLocation().getId());
            ps.setInt(6, item.getId());

            ps.executeUpdate();
        }
    }

    public void deleteItem(int itemId) throws SQLException {
        String sql = "DELETE FROM items WHERE id = ?";

        try (Connection conn = getConnection();
             PreparedStatement ps = conn.prepareStatement(sql))
{

            ps.setInt(1, itemId);
            ps.executeUpdate();
        }
```

```
    }
    // ==============================
    // Helper: load everything into Inventory
    // ==============================

    public void loadAllIntoInventory(Inventory inventory) {
        inventory.clear();  // this now calls Inventory.clear()

        List<Location> locations = loadLocations();
        List<Category> categories = loadCategories();

        for (Location loc : locations) {
            List<FoodItem> itemsForLoc =
loadItemsForLocation(loc, categories, locations);
            for (FoodItem item : itemsForLoc) {
                inventory.addItem(item);
            }
        }
    }

}
```

CategoryController.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.controller;

import com.james.groceryinventorymanager.dao.DatabaseAdapter;
import com.james.groceryinventorymanager.model.Category;
import com.james.groceryinventorymanager.model.Inventory;

import java.sql.SQLException;
import java.util.List;

/**
 *
 * @author james
 */

public class CategoryController {
```

```
    private final DatabaseAdapter db;
    private final Inventory inventory; // not used now, but
available if needed later

    public CategoryController(DatabaseAdapter db, Inventory
inventory) {
        this.db = db;
        this.inventory = inventory;
    }

    public List<Category> getAllCategories() {
        return db.loadCategories();
    }

    public void addCategory(String name) throws SQLException {
        Category c = new Category(0, name);
        db.insertCategory(c);
    }

    public void updateCategory(Category category, String
newName) throws SQLException {
        category.setName(newName);
        db.updateCategory(category);
    }

    public void deleteCategory(Category category) throws
SQLException {
        db.deleteCategory(category.getId());
    }
}
```

InventoryController.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.controller;

import com.james.groceryinventorymanager.dao.DatabaseAdapter;
import com.james.groceryinventorymanager.model.Category;
```

```java
import com.james.groceryinventorymanager.model.FoodItem;
import com.james.groceryinventorymanager.model.Inventory;
import com.james.groceryinventorymanager.model.Location;

import java.sql.SQLException;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author james
 */

public class InventoryController {

    private final Inventory inventory;
    private final DatabaseAdapter dbAdapter;

    private final LocationController locationController;
    private final CategoryController categoryController;

    private List<Location> locationsCache = new ArrayList<>();
    private List<Category> categoriesCache = new ArrayList<>();

    public InventoryController(Inventory inventory,
DatabaseAdapter dbAdapter) {
        this.inventory = inventory;
        this.dbAdapter = dbAdapter;

        this.locationController = new
LocationController(dbAdapter, inventory);
        this.categoryController = new
CategoryController(dbAdapter, inventory);

        reloadAll();
    }

    public LocationController getLocationController() {
        return locationController;
    }

    public CategoryController getCategoryController() {
        return categoryController;
    }
```

```
    public void reloadAll() {
        locationsCache = dbAdapter.loadLocations();
        categoriesCache = dbAdapter.loadCategories();
        dbAdapter.loadAllIntoInventory(inventory);
    }

    public List<Location> getAllLocations() {
        return new ArrayList<>(locationsCache);
    }

    public List<Category> getAllCategories() {
        return new ArrayList<>(categoriesCache);
    }

    public List<FoodItem> getItemsForLocation(Location location)
{
        return inventory.getItemsByLocation(location);
    }

    public List<FoodItem> searchItemsByName(Location location,
String query) {
        List<FoodItem> itemsInLoc =
inventory.getItemsByLocation(location);
        String lower = query.toLowerCase();
        List<FoodItem> result = new ArrayList<>();
        for (FoodItem item : itemsInLoc) {
            if (item.getName() != null &&
item.getName().toLowerCase().contains(lower)) {
                result.add(item);
            }
        }
        return result;
    }

    public List<FoodItem> filterByFreshness(Location location,
                                            boolean
includeFresh,
                                            boolean
includeExpiringSoon,
                                            boolean
includeExpired) {
        List<FoodItem> allFiltered =
                inventory.filterByFreshness(includeFresh,
includeExpiringSoon, includeExpired);

        List<FoodItem> result = new ArrayList<>();
```

```
        for (FoodItem item : allFiltered) {
            if (item.getLocation() != null &&
item.getLocation().equals(location)) {
                result.add(item);
            }
        }
        return result;
    }

    public List<FoodItem> sortByName(List<FoodItem> source,
boolean ascending) {
        return inventory.sortByName(source, ascending);
    }

    public List<FoodItem> sortByCategory(List<FoodItem> source,
boolean ascending) {
        return inventory.sortByCategory(source, ascending);
    }

    public List<FoodItem> sortByQuantity(List<FoodItem> source,
boolean ascending) {
        return inventory.sortByQuantity(source, ascending);
    }

    public FoodItem addItem(String name,
                            int quantity,
                            LocalDate expiration,
                            Category category,
                            Location location) throws
SQLException {
        FoodItem item = new FoodItem(name, quantity, expiration,
category, location);
        dbAdapter.insertItem(item);
        inventory.addItem(item);
        return item;
    }

    public void updateItem(FoodItem item,
                           String name,
                           int quantity,
                           LocalDate expiration,
                           Category category,
                           Location location) throws
SQLException {

        item.setName(name);
```

```
        item.setQuantity(quantity);
        item.setExpirationDate(expiration);
        item.setCategory(category);
        item.setLocation(location);

        dbAdapter.updateItem(item);
    }

    public void deleteItem(FoodItem item) throws SQLException {
        dbAdapter.deleteItem(item.getId());
        inventory.removeItem(item);
    }
}
```

LocationController.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.controller;
import com.james.groceryinventorymanager.dao.DatabaseAdapter;
import com.james.groceryinventorymanager.model.Inventory;
import com.james.groceryinventorymanager.model.Location;

import java.sql.SQLException;
import java.util.List;
/**
 *
 * @author james
 */

public class LocationController {

    private final DatabaseAdapter db;
    private final Inventory inventory;

    public LocationController(DatabaseAdapter db, Inventory
inventory) {
        this.db = db;
        this.inventory = inventory;
```

```
    }

    public List<Location> getAllLocations() {
        return db.loadLocations();
    }

    public void addLocation(String name) {
        try {
            Location loc = new Location(0, name);
            db.insertLocation(loc);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void deleteLocation(Location location) {
        try {
            db.deleteLocation(location.getId());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

InventoryFreshnessFilterTest.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.model;

import org.junit.jupiter.api.Test;

import java.time.LocalDate;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

/**
 *
 * @author james
```

```
 */

public class InventoryFreshnessFilterTest {

    @Test
    public void testFreshIncludesNullExpiration() {
        Inventory inventory = new Inventory();
        Location pantry = new Location(1, "Pantry");
        Category canned = new Category(1, "Canned");

        // Null expiration - should be treated as "fresh"
        FoodItem noDateItem = new FoodItem(0, "Beans (no date)",
2, null, canned, pantry);

        // Far future date - more than 30 days from now
        LocalDate future = LocalDate.now().plusDays(90);
        FoodItem futureItem = new FoodItem(0, "Pasta 90d", 1,
future, canned, pantry);

        // Expiring soon - within 30 days
        LocalDate soon = LocalDate.now().plusDays(10);
        FoodItem soonItem = new FoodItem(0, "Milk 10d", 1, soon,
canned, pantry);

        // Expired - before today
        LocalDate past = LocalDate.now().minusDays(1);
        FoodItem expiredItem = new FoodItem(0, "Old bread", 1,
past, canned, pantry);

        inventory.addItem(noDateItem);
        inventory.addItem(futureItem);
        inventory.addItem(soonItem);
        inventory.addItem(expiredItem);

        // Fresh only - should include null expiration and > 30
days
        List<FoodItem> freshOnly =
inventory.filterByFreshness(true, false, false);
        assertTrue(freshOnly.contains(noDateItem), "Null
expiration should be counted as fresh");
        assertTrue(freshOnly.contains(futureItem), "Future item
> 30 days should be fresh");
        assertFalse(freshOnly.contains(soonItem), "Soon item
should not be in fresh only");
        assertFalse(freshOnly.contains(expiredItem), "Expired
item should not be in fresh only");
```

```
        // Expiring soon only
        List<FoodItem> soonOnly =
inventory.filterByFreshness(false, true, false);
        assertTrue(soonOnly.contains(soonItem), "Item within 30
days should be expiring soon");
        assertFalse(soonOnly.contains(futureItem));
        assertFalse(soonOnly.contains(expiredItem));
        assertFalse(soonOnly.contains(noDateItem));

        // Expired only
        List<FoodItem> expiredOnly =
inventory.filterByFreshness(false, false, true);
        assertTrue(expiredOnly.contains(expiredItem), "Past date
should be expired");
        assertFalse(expiredOnly.contains(soonItem));
        assertFalse(expiredOnly.contains(futureItem));
        assertFalse(expiredOnly.contains(noDateItem));

        // All filters on - everything should appear
        List<FoodItem> all = inventory.filterByFreshness(true,
true, true);
        assertEquals(4, all.size(), "All four items should be
included when all filters are on");
    }
}
```

InventorySortingTest.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.model;
import org.junit.jupiter.api.Test;

import java.time.LocalDate;
import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;
```

```java
/**
 *
 * @author james
 */

public class InventorySortingTest {

    @Test
    public void testSortByNameAscendingAndDescending() {
        Inventory inventory = new Inventory();
        Location pantry = new Location(1, "Pantry");

        FoodItem c = new FoodItem(0, "Carrots", 3, null, null,
pantry);
        FoodItem a = new FoodItem(0, "Apples", 5, null, null,
pantry);
        FoodItem b = new FoodItem(0, "Bananas", 2, null, null,
pantry);

        inventory.addItem(c);
        inventory.addItem(a);
        inventory.addItem(b);

        List<FoodItem> source = Arrays.asList(c, a, b);

        List<FoodItem> ascending = inventory.sortByName(source,
true);
        assertEquals(Arrays.asList(a, b, c), ascending, "Sort by
name ascending should be A, B, C");

        List<FoodItem> descending = inventory.sortByName(source,
false);
        assertEquals(Arrays.asList(c, b, a), descending, "Sort
by name descending should be C, B, A");
    }

    @Test
    public void testSortByCategoryName() {
        Inventory inventory = new Inventory();
        Location pantry = new Location(1, "Pantry");

        Category dairy = new Category(1, "Dairy");
        Category canned = new Category(2, "Canned");
        Category snacks = new Category(3, "Snacks");
```

```
        FoodItem milk = new FoodItem(0, "Milk", 1, null, dairy,
pantry);
        FoodItem beans = new FoodItem(0, "Beans", 4, null,
canned, pantry);
        FoodItem chips = new FoodItem(0, "Chips", 2, null,
snacks, pantry);

        inventory.addItem(milk);
        inventory.addItem(beans);
        inventory.addItem(chips);

        List<FoodItem> source = Arrays.asList(milk, beans,
chips);

        List<FoodItem> ascending =
inventory.sortByCategory(source, true);
        // Alphabetical by category name: "Canned", "Dairy",
"Snacks"
        assertEquals(Arrays.asList(beans, milk, chips),
ascending);

        List<FoodItem> descending =
inventory.sortByCategory(source, false);
        assertEquals(Arrays.asList(chips, milk, beans),
descending);
    }

    @Test
    public void testSortByQuantity() {
        Inventory inventory = new Inventory();
        Location pantry = new Location(1, "Pantry");

        FoodItem low = new FoodItem(0, "Low", 1, null, null,
pantry);
        FoodItem mid = new FoodItem(0, "Mid", 5, null, null,
pantry);
        FoodItem high = new FoodItem(0, "High", 10, null, null,
pantry);

        inventory.addItem(low);
        inventory.addItem(mid);
        inventory.addItem(high);

        List<FoodItem> source = Arrays.asList(mid, high, low);
```

```
        List<FoodItem> ascending =
inventory.sortByQuantity(source, true);
        assertEquals(Arrays.asList(low, mid, high), ascending,
"Ascending should be 1, 5, 10");

        List<FoodItem> descending =
inventory.sortByQuantity(source, false);
        assertEquals(Arrays.asList(high, mid, low), descending,
"Descending should be 10, 5, 1");
    }
}
```

DatabaseAdapterTest.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/
license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/
Class.java to edit this template
 */
package com.james.groceryinventorymanager.dao;

import org.junit.jupiter.api.Test;

import java.nio.file.Files;
import java.nio.file.Path;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.fail;

public class DatabaseAdapterTest {

    @Test
    public void testInitializeDatabaseCreatesFile() {
        try {
            Path tempFile = Files.createTempFile("grocery-test-
db", ".sqlite");
            String dbPath =
tempFile.toAbsolutePath().toString();

            DatabaseAdapter adapter = new
DatabaseAdapter(dbPath);
            adapter.initializeDatabase();
```

```
            assertTrue(Files.exists(tempFile), "Database file
should exist after initialization");
        } catch (Exception e) {
            e.printStackTrace();
            fail("Database initialization should not throw: " +
e.getMessage());
        }
    }
}
```