RIPHAH INTERNATIONAL UNIVERSITY, GGC.

Faculty of Computing

Formal Methods in Software Engineering

Semester Project

Submitted To: Ma'am Kausar Nasreen

Submission Date: 15th May, 2025.

Full Name of Student	**CMS Numbers**	**Program**
Ayla Amir	35702	BSSE
Alishba Sajid	44517	BSSE
Ayesha Asad	44587	BSSE
Manahil Habib	47876	BSSE
Nagarash Fateh	44815	BSSE

Final Report: Integration of Formal and Informal Methods in Software Development

Case Study: GrocerStore – A Console-Based Grocery Management System

1. Introduction

In today's world, software development requires a careful balance between being flexible and ensuring reliability. Developers often use informal techniques like user stories, flowcharts, and iterative prototyping to quickly adapt to user needs. However, these methods can sometimes struggle to provide the level of correctness, consistency, and safety that's essential. That's where Formal Methods (FM) come into play—these are a set of mathematically rigorous techniques that are crucial for ensuring quality.

This report showcases how both informal and formal methods are blended together in the creation of GrocerStore, a C++ console-based grocery management system. Crafted by a team of five, this project mimics a real-world shopping experience while incorporating both user-friendly features and behaviors that have been formally verified.

2. Project Overview

- **Project Purpose**
  The purpose of GrocerStore is to create a console-based system to manage various aspects of grocery store operations. The project aims to apply **Object-Oriented Programming (OOP)** principles such as encapsulation, inheritance, and polymorphism to build a structured and scalable application.
- **Architecture and Major Component**

  The system follows a modular, object-oriented design. Each key operation is encapsulated in its own class or module, promoting **separation of concerns** and **code reusability**.

  - **main.cpp:** Acts as the entry point and control hub.
  - **User.cpp /User.h:** Handles signup, login, and user roles.
  - **Admin, Manager, Customer:** Implement role-specific interfaces and privileges.
  - **Inventory, Catalog, AddCart, Checkout:** Handle product display, cart updates, and final billing.
  - **Feedback:** Collects customer reviews.

  This architecture supports scalability and makes the system easy to test and maintain.

- **Technologies Used**
  The system uses following technology:
  - **Programming Language**: C++
  - **Programming Concepts**: OOP, file handling, modular architecture
  - **Development Paradigm**: CLI-based system using standard I/O

Key Functionalities

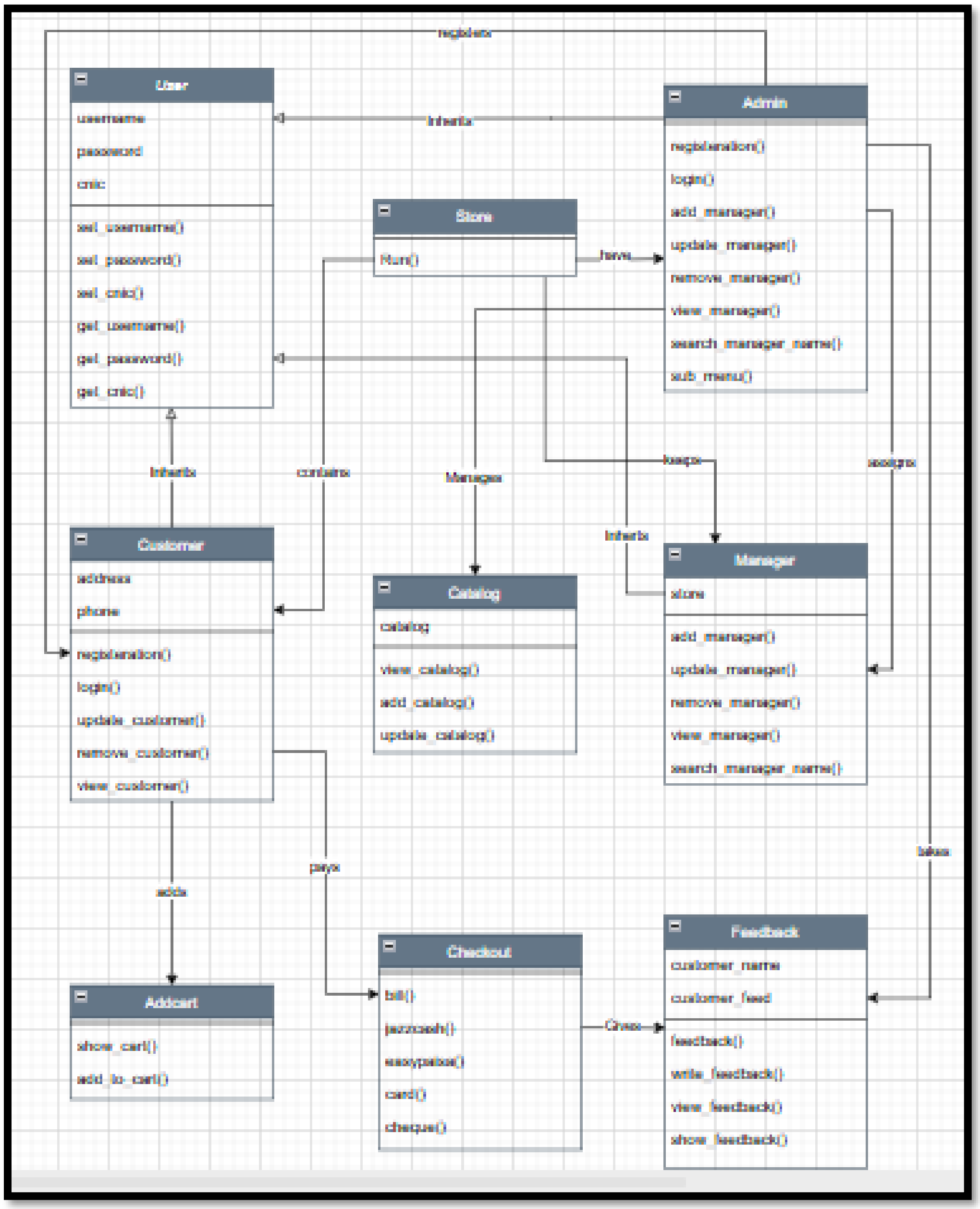
- Secure user registration and login
- Inventory management and product catalog
- Role-based access control
- Shopping cart and checkout process
- Customer feedback collection

System Flow Summary

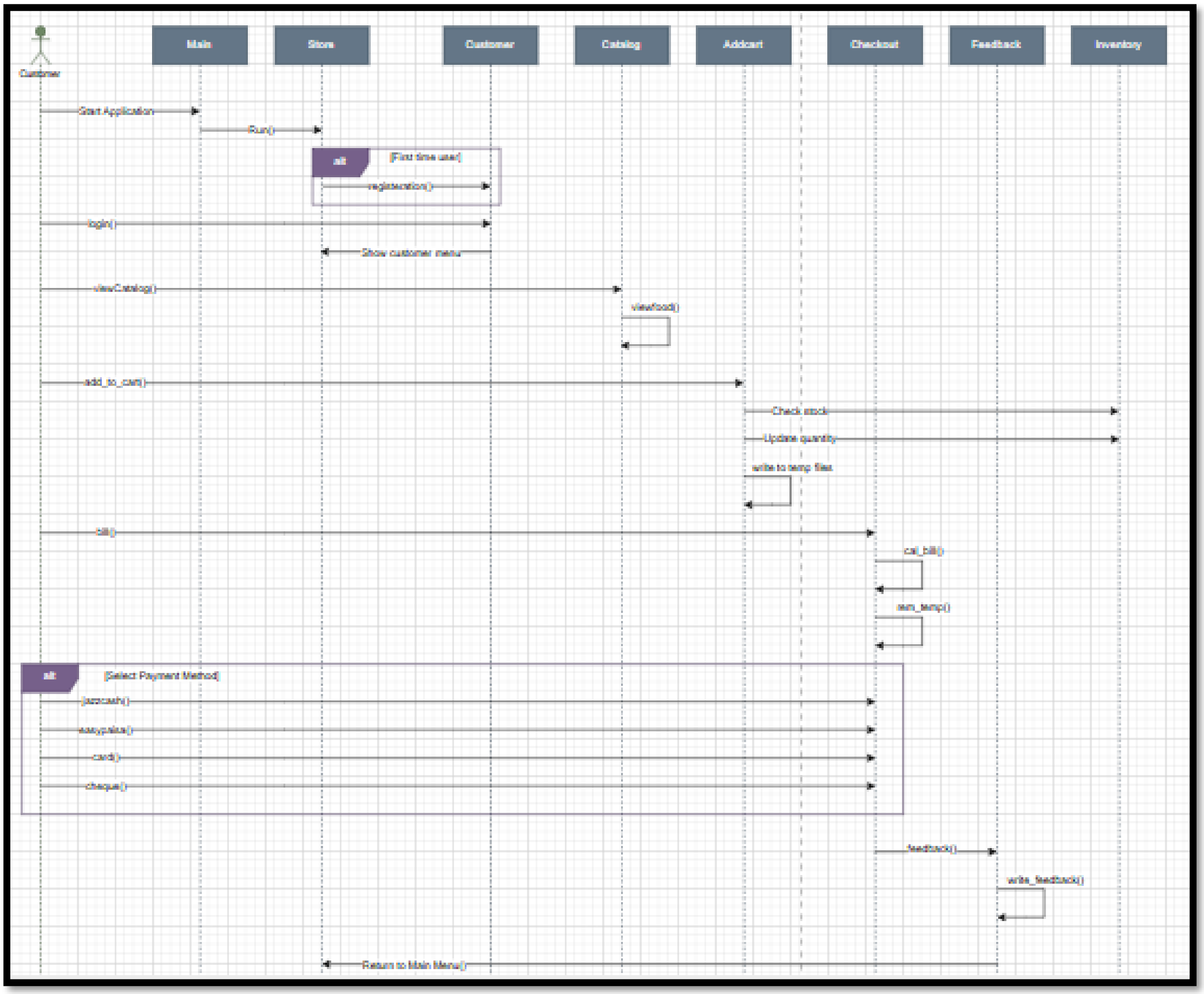
- Program launches from `main.cpp`.
- User signs up or logs in.
- Role is determined (Admin, Manager, or Customer).
- Role-specific functionalities are made available.
- Customer can browse catalog, add to cart, and checkout.
- Admin/Manager can add/view inventory, view transactions.

3. UML Diagrams and Their Relevance

- **Class Diagram**
  - **Purpose**: Visualizes the static structure of the system.
  - **Usage**: Shows relationships such as inheritance (`Admin` inherits from `User`), composition (`Cart` contains `Items`), and associations.
  - **Relevance**: Helps understand code organization and dependencies.
- **Class Diagram Link**
  - https://app.diagrams.net/#G1rm9uFOBtr6WblnCNl0cbbOGxZudvtBfN#%7B%22pageId%22%3A%22C5RBs43oDa-KdzZeNtuy%22%7D

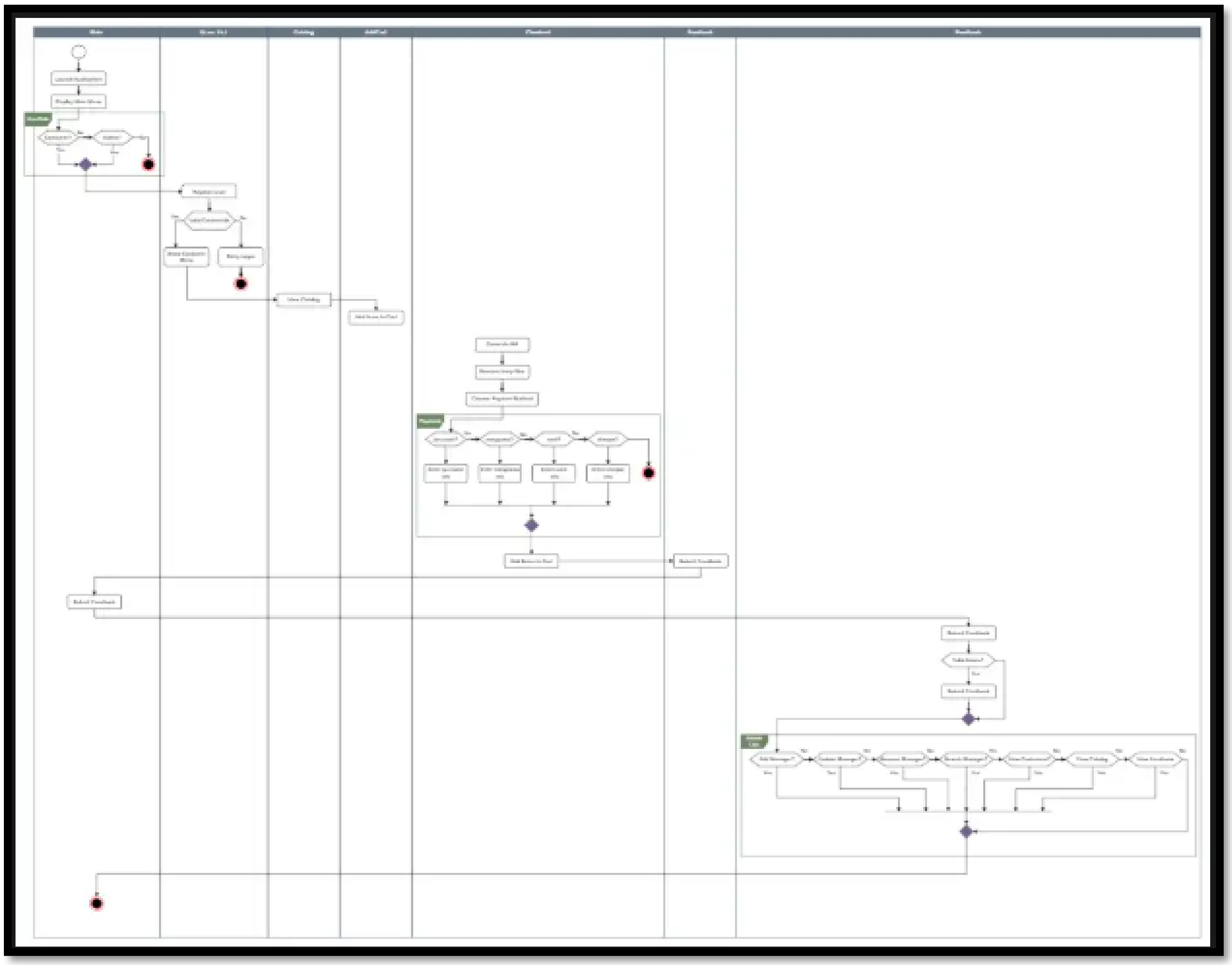


- **Sequence Diagram**
  - **Purpose**: Illustrate interaction between objects over time.
  - **Use Cases**: "User logs in", "Customer checks out", etc.
  - **Relevance**: Clarifies flow of messages and method calls during execution.
- **Sequence Diagram Link**
  - https://app.diagrams.net/#G1m5ytNX-YE7jvreMlWVkHfX_bR1WrQanN#%7B%22pageId%22%3A%222YBvvXClWsGukQMizWep%22%7D



Illustrates the **runtime interaction** between user and system during core use cases like **login**, **add to cart**, or **checkout**. Helps understand the object-to-object message flow.

- **Activity Diagram**
  - **Purpose**: Model dynamic workflows and decision logic.
  - **Use Cases**: Shopping process, inventory update, payment flow.
  - **Relevance**: Helps map informal workflows into formal logic for later validation.
- **Activity Diagram Link**
  - https://app.diagrams.net/#G1QTxit6Tc2gxVJ_OPauaLAJMcmCfsSAZn#%7B%22pageId%22%3A%22prtHgNgQTEPvFCAcTncT%22%7D



Describes the **workflow or control flow** of processes such as inventory updates, cart addition, or payment. Highlights decision-making logic and parallel actions.

- **Why UML is Important**

  - Clarifies software design
  - Helps translate informal logic to formal structure
  - Aids in identifying potential system flaws early in design

4. Formal Methods Analysis

Z-Notation (Informal Mapping)

Z-Notation is used to define **state transitions** and **data integrity rules**. While not natively supported in C++, schemas were manually modeled to describe:

- **Inventory Management**:
  - `AddItem`: Add new item to the product list.
  - `UpdateQuantity`: Reduce quantity after purchase.
- **Cart Operations**:
  - Maintain price and quantity sequences.
- **Password Validation**:
  - Schema to enforce rules: contains uppercase, lowercase, digits, special characters, and at least 8 characters.
- **Inventory Consistency**:
  - Ensures prices, items, and quantities remain aligned.

2. Algebraic Specification

Used to represent operations like:

- `add_to_cart(q, i, Δ)` → Reduces item quantity
- `calculate_total(p, q)` → Calculates bill
- `view_items(is, qs, ps)` → Combines inventory display

These were mapped using **input/output relationships and override functions**, ensuring correctness of computation.

3. Model Checking

Key behaviors were formally verified using model checking principles:

- Menu flows always reach checkout (liveness)
- Cart quantities never go negative (safety)
- Role-based login always leads to proper UI flow
- Payment options are always reachable

5. Logical Validation with Truth Tables

The code includes several conditionals (`if-else`, `switch-case`) for login checks, menu navigation, and payment selection. Truth tables were used to validate these:

Condition	Login Input	Expected Output
valid username/password	true	login successful
invalid credentials	false	retry or redirect

Similar truth tables were created for:

- Role-based access (admin vs customer)
- Cart quantity updates (`if quantity > 0`)
- Checkout conditions (selecting payment method)

These validations confirmed that **logic was sound** and covered all meaningful paths.

TRUTH TABLES

1. **Menu Navigation Logic (e.g., Inventory::addinven, viewinven, Addcart::showinven)**

CODE:

```
if (in == 1) { addfood(); }

else if (in == 2) { addhygene(); }

else if (in == 3) { addhouse(); }

else { return true; }
```

LOGIC:

The system should take an action if one of the expected options (1, 2, 3) is selected.

TRUTH TABLE:

Ensures the user selected a valid menu option.

Accepts in == 1, in == 2, or in == 3.

Invalid input leads to an exit or return.

In == 1	In == 2	In == 3	Action Performed
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

LOGICAL CORRECTNESS: The logic accepts at least one correct input and takes appropriate action. It only fails (returns true) when all are 0.

2. Password Validity Logic (in Manager::registeration)

CODE:

if (lowercase && uppercase && digit && specialchar && eightchars) { ... }

LOGIC:

Password must satisfy **all five conditions** to be valid.

TRUTH TABLE:

Requires the following conditions to all be true:

- Exactly 8 characters
- At least 1 lowercase letter
- At least 1 uppercase letter
- At least 1 digit
- At least 1 special character

Implemented using boolean flags and if conditions**.**

8 Chars	**Lower Case**	**Upper Case**	**Digit**	**Special Char**	**Valid Pass**
0	1	1	1	1	0
1	1	1	1	1	1
others	Any	any	any	any	0

LOGICAL CORRECTNESS: The truth table confirms that only if all flags are set, the password is accepted. Otherwise, it is rejected.

3. Quantity Validation for Adding to Cart (in addcmeat, addchygene, etc.)

CODE:

```
if (quant <= arr[in - 1]) {
    arr[in - 1] -= quant;
}
```

LOGIC:

Only allow subtraction from stock if there’s enough quantity.

TRUTH TABLE:

Ensures user doesn't add more items than are available in stock.

Two required conditions:

- The quantity in inventory (arr[i]) is ≥ the requested amount.
- The item index is valid.

Qty > △	**Valid Index**	**Allow Action**
0	0	0
0	1	0
1	0	0
1	1	1

LOGICAL CORRECTNESS: Both conditions must be true to perform subtraction. This prevents invalid memory access or negative stock.

OVERVIEW:

Truth Table Name	**Condition Modeled**
Menu Option Logic	Validates user input for selecting inventory category (1,2,3)
Password Validation	Validates password against 5 required criteria
Quantity Availability Check	Ensures selected item quantity does not exceed available stock

6. Integration of Formal and Informal Methods

◆ Informal Methods Used:

- Requirement gathering from users
- Designing class structure in C++
- Implementing console-based logic using if-else, loops
- Manually testing user flows

◆ Formal Methods Used:

- Z-Notation to define data transitions
- Algebraic specs to define pure operations
- Model checking to validate navigation, billing, and authentication logic

◆ How They Complement Each Other in SDLC:

Phase	Informal	Formal	Benefit
Requirements	User needs, roles	–	Understand what to build
Design	UML, flowcharts	Z-schemas	Clarify structure + logic
Development	C++ OOP	Algebraic specs	Implement with mathematical correctness
Testing	Manual testing	Model checking	Catch edge cases & verify behavior
Maintenance	Bug fixing	Re-validating logic	Ensure system consistency

By combining both approaches, the system became **robust, user-friendly, and logically sound**.

7. Conclusion

The GrocerStore project has proven that **integrating informal and formal methods** leads to software that is not only functionally complete but also **structurally sound and mathematically verified**.

Informal methods allowed quick prototyping and understanding of real-world user needs. Formal methods, on the other hand, helped ensure that **critical logic, data transitions, and workflows were verifiable and correct**.

This hybrid approach provided a **strong foundation for quality assurance, maintainability, and user trust**, making GrocerStore a well-rounded software development case study.