FINAL PROJECT

A Report
Presented to
School of Engineering and Computer School
Concordia University

In Fulfillment
of the Requirements
of COMP 354

By
George Kerasias (40283826)
Pierre-Olivier Gattillo (40228938)
Andy LeytonYew-Hin (40274109)
Armen Jabamikos (40273685)
Mattis Wan-Bok-Nale (40274067)
Geon Kim (40264507)
Concordia University

Table of Contents

1. Executive Summary

Across **Assignments 1, 2, and 3**, the **MyBankUML** system evolved into a multi-layered banking application that culminates in this final project: the implementation of a fully functional software system with a graphical user interface (**GUI**). The overall purpose of this project is to transform the architectural and design models created earlier in the term into a complete, executable application that integrates all major functional and non-functional requirements:

- **FR-1 – Account viewing**
- **FR-2 – Role-based access control (RBAC)**
- **FR-3 – Multi-filter account search**

At the top of the semester, **Assignment 1** established the **Software Requirements Specification (SRS)**, defining the system's functional requirements, non-functional constraints, usability targets, and performance expectations. After that, **Assignment 2** translated those specific requirements into architectural and component-level designs, and also specified the controllers, services, repositories, domain models, and system interactions through detailed sequence diagrams (**UC-01** and **UC-02**). This work finally culminated in **Assignment 3**, which followed the design by producing component tests, integration plans, traceability matrices, and design-to-requirement correlations.

Now, this final project executes everything stated above as a complete system that uses those deliverables as its blueprint. The implementation includes a fully interactive **GUI**, an integrated backend based on the A2 architecture, a centralized **RBAC** pipeline, a performance-tested multi-filter search, and end-to-end integration of controllers, services, and repositories. All of these components were implemented following a structured development plan, with team roles clearly defined and risks continuously identified and mitigated throughout the execution phase.

The purpose of this report overall is to document the **planning**, **execution**, **risk management**, and **implementation** leading into the final integration of the **MyBankUML** application. It presents the complete development timeline, the responsibilities of all team members, the feasibility of the project plan, the materialized risks and their mitigation, as well as a detailed account of how each module was coded, tested, and integrated to maintain alignment with the **SRS**, **A2** architecture, and **A3** testing outcomes. Together, this provides a comprehensive view of the software engineering lifecycle followed in **COMP 354**, from requirements to design, implementation, testing, and delivery.

2. Introduction

2.1 Purpose of Project

The original purpose of this project was defined, and iteratively redefined, across **Assignments 1, 2, and 3**. While the assignments themselves focused on **requirements**, **architecture**, and **integration planning**, this final phase transforms that design into a fully functioning software product that includes a complete **Graphical User Interface (GUI)**.

The purpose of this project is to implement the **complete and functional version of the MyBankUML application**, integrating all specifications defined in the earlier stages of the coursework. Building on the requirements from **Assignment 1**, the architectural blueprint from **Assignment 2**, and the integration strategy from **Assignment 3**, this final implementation brings the system to life by:

- **Implementing a fully operational Graphical User Interface (GUI)** for Customers, Tellers, and Admins.
- **Completing all backend components** — including SearchService, AccountRepository, AuthzService, RoleAdminController, MaskingPolicy, and user-role persistence.
- **Ensuring the system enforces all RBAC rules**, masking policies, and usability constraints (e.g., Teller search flow must be ≤ 3 actions).
- **Executing the implementation following a structured project plan**, supported by scheduling, risk analysis, mitigation strategies, and feasibility assessments.

In essence, this project represents the transition from **design** to **deployment**, ultimately demonstrating the team's ability to engineer a complete software solution grounded in well-defined specifications, architecture, testing deliverables, and GUI-driven usability expectations

2.2 Scope

The **scope of the MyBankUML project** implements two key deployment scopes.

Technical Deployment Scope

This scope covers everything related to **implementation**, **system behaviour**, and **technical feasibility**. It includes:

- **Graphical User Interface (GUI)**
  Developed using Java (Swing/JavaFX), featuring separate views for **Customers**, **Tellers**, and **Admins**.
- **Role-Based Access Control (RBAC)**
  Enforced through **AuthzService**, **MaskingPolicy**, and **RoleAdminController**, ensuring secure interactions aligned with **Assignment 1** requirements.
- **Search and Filtering**
  Multi-filter, **AND-based** search with pagination and performance constraints (≤ **2 seconds** for **10,000 accounts**).

- **Repository and Data Layer**
  A fully implemented **AccountRepository** with indexed search and a generated dataset for performance testing and validation.
- **System Integration**
  All controllers, services, repositories, and RBAC components integrated into a **single executable application**.
- **Testing and Validation**
  Includes component tests (from **A3**), integration tests (**IT-01 to IT-05**), performance checks, and GUI-level usability validation.

Project Management Scope

The second scope covers all responsibilities related to **project planning**, **execution**, and **team coordination**. This includes the overarching strategy and documentation required by the course.

Project management activities include:

- **A feasible and documented project schedule**
- **Risk identification and mitigation plans**, including critical and catastrophic risks
- **Execution review**, documenting materialized risks and how they were handled
- **Contribution tracking** and final verification (**QA**)
- **Feasibility assessment**, ensuring the plan could be completed within the timeline

2.3 Link to Assignments A1 → A2 → A3 → Project

This project does not stand alone as its own work. It is the combination of a **structured software deployment life cycle** that was executed across the semester. In order to situate this final work within that trajectory, it is important to start from the beginning and trace how each assignment directly influenced this project.

Assignment 1 (Requirements Specification)

- Defined the system's functional requirements (**FR-1**, **FR-2**, **FR-3**) and non-functional requirements (**performance**, **usability**, **security**).
- These requirements became the foundation for the architecture and all code developed in this project.

Assignment 2 (Architectural & Component Design)

- Produced the layered architecture (**UI → Controllers → Services → Repositories → Domain**), component descriptions, and two sequence diagrams (**UC-01** and **UC-02**).

- This design directly shaped the implementation choices and the code structure.

Assignment 3 (Integration Strategy & Testing)

- Verified the alignment between A1 requirements and A2 design, identified mismatches, and produced the **integration testing plan**, component tests, RBAC tests, and **traceability matrices**.
- All A3 test IDs (**CT-XX**, **IT-XX**) appeared in the development of this project.

Final Project (Implementation + GUI + Planning)

Implements the complete **MyBankUML system**, incorporates the GUI elements, and executes the project plan stated in the previous assignments.

Thus, this final project is a continuation of the earlier assignments, demonstrating how the **requirements evolve into design**, how **design evolves into implementation**, and finally how **implementation is validated** through structured testing and project-management practices. The next sections summarize the requirements that guided this project.

3. Summary Requirements

It is important to start at the core of the system requirements that guided all design and development throughout the semester. These were originally defined in **Assignment 1 (SRS)**, refined in **Assignment 2 (architecture & design)**, and validated through testing in **Assignment 3**. Together, these three form the **baseline of the GUI-enabled system**, and the subsections below summarize the functional and non-functional requirements to show the **full traceability** from the original SRS to the final implemented software demonstrated in the demo.

3.1 Functional Requirements

The MyBankUML system is built around three central functional requirements. All design decisions in A2, test cases in A3, and final project implementation map directly to these FRs.

FR-1 — Display Account Information

- The system must allow users to view bank account information securely.
- Customers may view **only their own accounts**, whereas Teller and Admin roles may view any customer's account.
- Displayed information includes account type (Card/Saving/Check), masked account number, and appropriate balance visibility.
- All visibility rules depend on **role-based restrictions** (implemented via AuthzService & MaskingPolicy).

FR-2 — Role-Based Permissions (RBAC)

- The system must enforce strict authorization rules for all operations.
- Customers: view their own accounts only.
- Tellers: search and view customer accounts but may *not* manage roles.
- Admins: full visibility + ability to assign/remove roles.
- Sensitive fields (e.g., full account number, balance) must be masked unless the user's role permits access.
- Role changes must take effect **immediately** (validated in A3 TR-01).

FR-3 — Search Accounts

- Teller and Admin users must be able to search accounts using **one or more filters** (account number, type, customer name).
- Results must respect RBAC constraints: Customer queries automatically restricted to only their accounts.
- Search must support **pagination (≤ 50 results)** and **multi-filter AND logic** (A3 clarification TR-02).

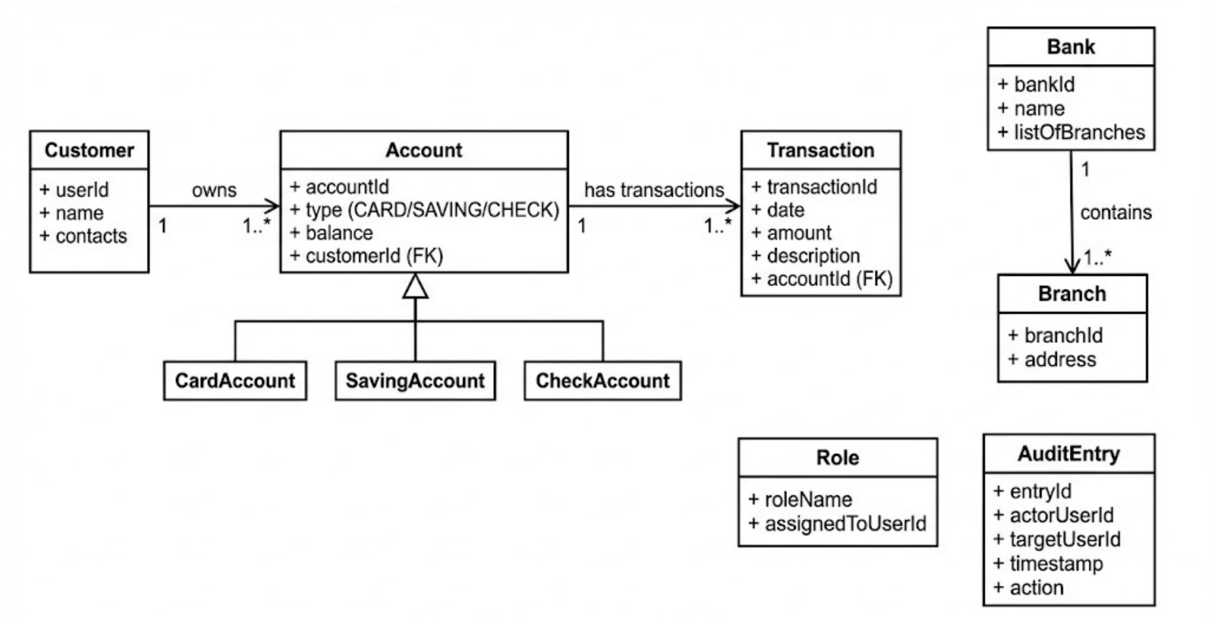
- If no results match, the system must display a clear empty-result message.

Table 3.1 — Functional Requirements Summary

ID	Description	Implemented In	Verified In
FR-1	System shall display all accounts belonging to the logged-in Customer; Teller/Admin can view any account.	AccountViewController, AccountQueryService, AuthzService	CT-AVC, CT-AQS, IT-01
FR-2	Role-based access (RBAC): restrict Customer views; allow Teller/Admin to view/search; masking of sensitive data; role assignment/removal.	AuthzService, MaskingPolicy, RoleAdminController, RoleRepository	CT-AUTH, CT-MASK, IT-03, IT-04
FR-3	Multi-filter search on accounts (ID, customer name, type), pagination, AND-combination of filters; ≤2s for 10k accounts.	SearchService, SearchController, AccountRepository	CT-SS, CT-AR, IT-02

To contextualize the functional requirements and show the underlying structure of the system specified in Assignment 1, Figure 3.1 reproduces the original MyBankUML domain model.

Figure 3.1 — MyBankUML Domain Model



3.2 Non-Functional Requirements

The system's quality attributes were established in A1 and verified in A3. The final implementation must adhere to the following:

NFR-1: Security

- RBAC must be strictly enforced across all components.
- Unauthorized access must result in immediate rejection with an audit log entry (A3 IT-02, CT-A4).
- Masking rules must be applied consistently across all views and search results.

NFR-2: Usability

- Teller workflow must allow searching → selecting → opening an account in **≤ 3 user actions**, as defined in A1.
- GUI must clearly differentiate roles and prevent user confusion.
- Error messages must be actionable and consistent.

NFR-3: Performance

- The system must return search results for datasets up to **10,000 accounts within 2 seconds** (validated in A3; refined in TR-03).
- *Account details must load within **1 second**.*

NFR-4: Auditability

- All role changes must be recorded with admin ID, target user, role added/removed, and timestamp (TR-04).
- Denied access attempts must also generate an audit entry.
- Logs must be consistent and persist through system restarts.

Now that the **non-functional** and **functional** requirements have been clearly laid out, the next section will describe how the **software development plan** guided our implementation. This will include the **work breakdown structure**, the team, the **roles of each team member**, the **project schedule**, the **feasibility analysis** conducted, and the **planning methodology** that ensured that the MyBankUML system could be delivered on time and according to the requirements set out in all assignments.

<u>4. Software Plan</u>

Moving on, now that the established functional and non-functional requirements in **A1** have been completed, along with the architecture and component design in **A2**, and the integration and test strategy in **A3**, the next step is to define the **concrete software development plan** for this final project. Thus, this section will describe how the team planned and organized the crucial steps to implement the **MyBankUML** application with a graphical interface. This will include the **task breakdown**, **role assignment**, **scheduling**, and **feasibility**.

4.1 Planning Approach

To develop the **MyBankUML application** with a GUI, the team followed an **iterative process** that was deliberately aligned with the earlier assignments. To begin, there was the **requirements-driven planning**. The scope of the work here was derived directly from **A1** (**FR-1 Display Account Info, FR-2 RBAC, FR-3 Search Accounts**, and the **NFRs** on security, usability, and performance). The plan ensured that every major requirement from A1 had at least one corresponding implementation task and one test activity.

Architecture-informed structure

The layered architecture and component decomposition proposed in **A2** (controllers, services, repositories, RBAC, masking, and future GUI) were used as the backbone of the work plan. Each major component from A2 became a unit of work in the project plan, which made planning traceable and reduced the risk of “orphan” code that did not map to a requirement.

Iterative development with parallel work streams

To make effective use of the team of six, we organized the work into parallel streams:

- GUI/UX implementation

- Search and performance
- Repository and data layer
- RBAC, masking, and audit logging
- System integration
- Project management, QA, and final report

Each stream iterated over **design → implementation → local testing**, then handed off to the **Integration Engineer** for end-to-end validation.

Feature branches and controlled integration

All developers worked on feature branches (e.g., **feature/gui**, **feature/search**, **feature/rbac**, **feature/repository**) and merged into the shared main branch only after local compilation and basic tests. This aligned directly with the **top-down integration strategy** defined in A3, where controllers are integrated first, then services, then repositories and RBAC.

Top-down integration and code freeze

The plan adopted the **top-down integration process**, which is consistent with **Assignment 3**. Here, there was an assurance that the high-level flows, such as **view account**, **search**, and **manage roles**, were wired through the controllers and services first, and then we progressively connected the **repository** and **RBAC** components. This ensured stability before the final report. A **hard code freeze** was implemented for **November 20**, which allowed a dedicated window of five days for the final questions, final QA, and report writing.

Overall, the planning here emphasized **traceability**, **parallelism**, and **controlled integration**, and used **Assignment 1**, **Assignment 2**, and **Assignment 3** as consistent anchors throughout the duration of the project to re-center the approach.

4.2 Work Breakdown Structure

The **Work Breakdown Structure (WBS)** decomposes this project into manageable units that were aligned with the architectural requirements. Each of the smaller work packages maps to one or more components from **Assignment 2** and to the test suites from **Assignment 3**.

Level 1 – Project MyBankUML with GUI

- **W1 – GUI Layer (FR-1, FR-2, FR-3 + NFR-Usability)**
    - **W1.1:** Set up base GUI framework (Swing/JavaFX, main window, navigation)
    - **W1.2:** Implement Customer account view screen (view own accounts, masked fields)

    - **W1.3:** Implement Teller search screen (filters, pagination, results table)
    - **W1.4:** Implement Admin role management screen (assign/remove roles, list roles)
    - **W1.5:** Connect GUI events to controllers (AccountViewController, SearchController, RoleAdminController)
    - **W1.6:** Usability refinement to meet ≤ 3 actions for Teller search → results → detail
- **W2 – Backend Services (FR-1, FR-3)**
    - **W2.1:** Implement AccountQueryService with masking-aware AccountSummary
    - **W2.2:** Implement SearchService with multi-filter AND logic (TR-02)
    - **W2.3:** Implement pagination logic and DTO mappings for search results
    - **W2.4:** Implement performance measurement hooks (timers for search and view)
- **W3 – Repository and Data (FR-1, FR-3 + NFR-Performance)**
    - **W3.1:** Implement AccountRepository findById and filtered search with pagination
    - **W3.2:** Design and generate a 10,000-record account dataset for performance testing
    - **W3.3:** Implement a repeatable data loader (e.g., CSV or script) for the repository
    - **W3.4:** Validate queries and verify index usage for performance targets
- **W4 – RBAC, Masking, and Audit (FR-2 + NFR-Security + Auditability)**
    - **W4.1:** Implement AuthzService with per-request role checks (TR-01)
    - **W4.2:** Implement MaskingPolicy (Customer vs Teller vs Admin visibility)
    - **W4.3:** Implement RoleRepository for persistent role storage
    - **W4.4:** Implement RoleAdminController endpoints for assign/remove/list roles
    - **W4.5:** Implement audit logging for role changes (fields from TR-04)
- **W5 – Integration and Build (All FRs/NFRs)**
    - **W5.1:** Configure Maven build, dependencies, and execution entry point

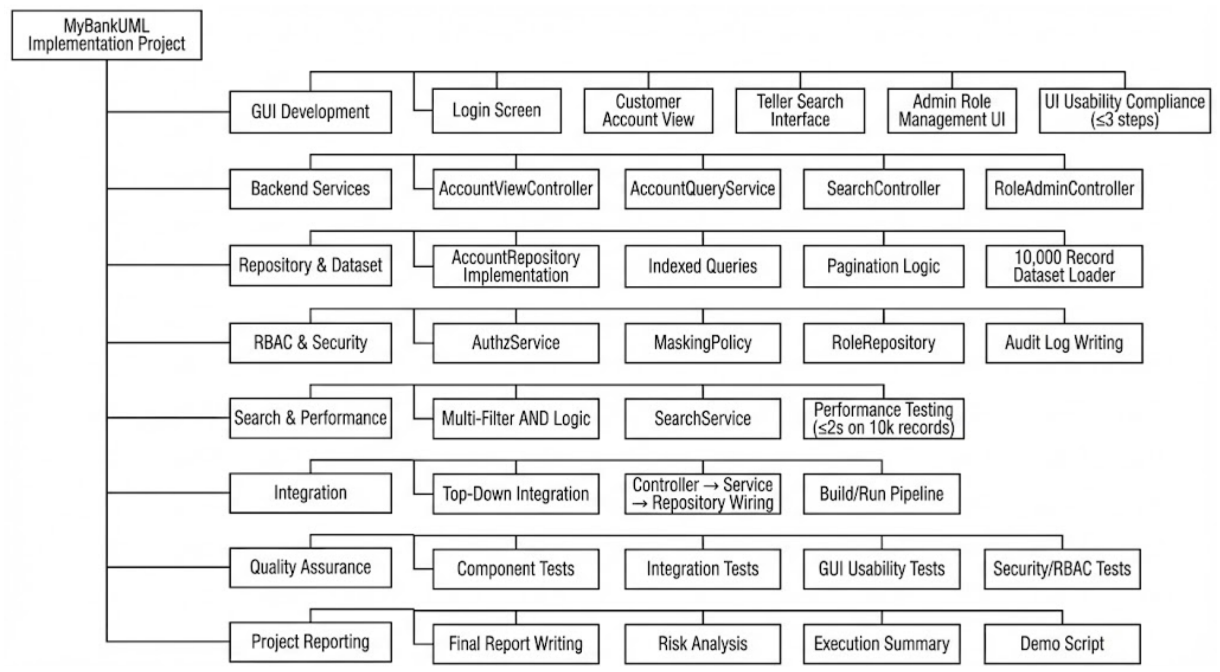
- **W5.2:** Integrate controllers ↔ services ↔ repositories ↔ RBAC
- **W5.3:** Run top-down integration tests (IT-01..IT-05 from A3)
- **W5.4:** Package runnable application and document run instructions

- **W6 – Quality Assurance and Reporting (Project-specific requirement)**
  - **W6.1:** Define and execute component tests (CT tables from A3)
  - **W6.2:** Run GUI usability checks against NFR-Usability (≤ 3 steps, masking visible)
  - **W6.3:** Conduct performance tests on the 10k dataset (NFR-Performance)
  - **W6.4:** Perform final QA pass and create demo walkthrough
  - **W6.5:** Write the final project report (planning, risks, implementation, tests, changes)

This **WBS** ensured that every delivery in the project description (plan, GUI, risk analysis, implementation, tests, and integration) had a clear outline and a responsible person.

This section decomposes the MyBankUML implementation project into major task groups and their subordinate deliverables. The WBS follows a hierarchical structure consistent with the planning approach defined in Section 4.1 and provides full visibility into parallel workstreams handled by the team.

Figure 4.2 — Work Breakdown Structure for MyBankUML Implementation



4.3 Roles and Responsibilities

To keep this final project manageable and aligned with each individual team member's strengths, the roles were assigned per **feature stream**. **George** acted as the **executive project manager** and the **main report writer**, while the other team members focused on specialized implementation tasks and integration. This approach allowed each person to concentrate on their individual responsibilities, while one person managed the overall workflow to ensure that all deadlines were being met.

Table 4.3 – Roles, Responsibilities, and Deliverables

Person	Role	Responsibilities	Deliverables by Nov 20
George	Project Manager + QA + Report Writer	Project planning and scheduling; risk analysis; feasibility assessment; final QA; integration validation; demo script and walkthrough; full report writing.	Project plan & schedule; risk analysis draft; QA sign-off notes; demo script; final report (Nov 20–25).
Pierre-Olivier	Search & Performance Lead	Implement FR-3 Search (SearchService + SearchController); multi-filter AND logic; pagination; performance tuning and evidence.	Fully working search backend wired to GUI; search DTOs; performance timings on 10k dataset; test notes.
Andy	GUI/UX Lead	Implement all GUI screens (Customer, Teller, Admin); integrate UI with controllers; enforce ≤3-step Teller flow; ensure masking is correctly rendered.	Complete GUI wired to backend; screenshots of all screens; click-flow verification (≤ 3 actions).
Armen	Repository/Data Engineer	Implement AccountRepository; design and load ~10k account dataset; support filtered queries and pagination; ensure query performance.	Functional repository layer; dataset loader/generator; verified query behavior and performance notes.
Mattis	Integration Engineer	Integrate controllers → services → repositories → RBAC; fix dependency mismatches; ensure IT-01..IT-05 from A3 pass in the running system; provide build instructions.	Stable, runnable application; integration logs; successful execution of key integration scenarios.
Geon	RBAC & Audit Engineer	Implement AuthzService, MaskingPolicy, RoleAdminController, RoleRepository; implement audit logging for role changes; ensure RBAC enforced in GUI.	Full RBAC pipeline; masking applied in DTOs and GUI; role management functioning; audit logs for changes.

In the Gantt-style view, **W1 through W5** were running in parallel during **November 10th to 20th**, with **W6** being the final report-writing session that was concentrated afterward. The **critical path** runs through the **repository + search + GUI + RBAC + integration**, and all of these had to be stable by **November 20th** so that the report work would not be blocked by any late code changes that were instrumental to the report.

4.4 Project Timeline

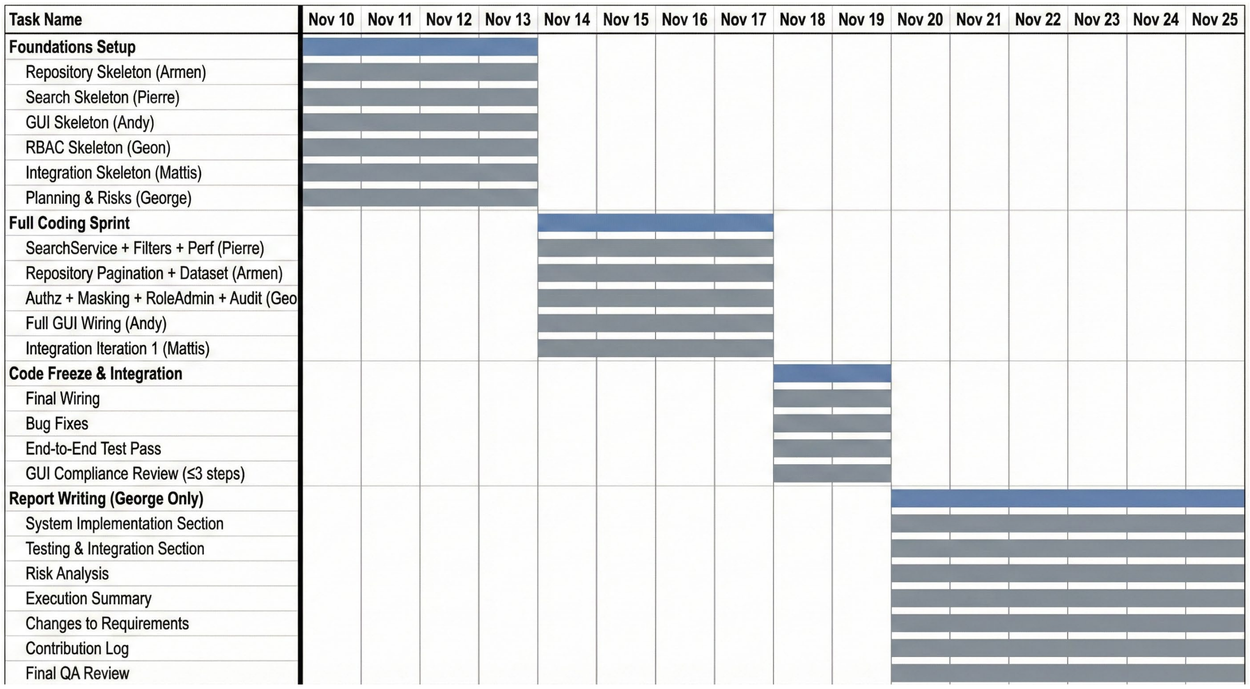
Once each responsibility was outlined per person, it was now time to create a **project timeline**. This timeline was organized around the course deadline (**November 27th**) and the **hard code freeze on November 20th**, which reserved sufficient time for final report writing and any final questions or issues. The following table shows the structure that was followed.

Table 4.4 – Project Timeline and Milestones

Period	Focus	Key Activities & Outputs
Nov 10–14	Foundations & Environment Setup	Maven/Git setup; base GUI skeleton; initial AccountRepository; SearchService and RBAC scaffolding; project plan, schedule, and initial risk list drafted by George.
Nov 15–18	Full Coding Sprint	GUI screens implemented; SearchService multi-filter logic; MaskingPolicy and AuthzService; repository pagination and dataset loading; integration pipeline built; George refines schedule and feasibility notes.
Nov 19–20	Code Freeze & Integration	**Nov 20 – Code Deadline:** all features implemented; controllers→services→repositories→RBAC fully wired; integration tests and smoke tests run; preliminary performance and usability checks performed.
Nov 20–25	Reporting & Final QA	George writes the entire report (planning, risks, implementation details, testing results, changes); collects screenshots and issue lists from teammates; final QA pass on the running app.
Nov 27	Submission & Demo	Final report submitted; demo prepared and rehearsed using the integrated build and demo script.

Figure 4.4 provides the project timeline in Gantt chart form. It visualizes the sequencing of major tasks, parallel development streams, the code freeze period, and the final report window.

Figure 4.4 — Project Timeline (Gantt Chart) from Nov 10–25



4.5 Feasibility Analysis

After deciding which members of the team were going to tackle which tasks, and when each task would take place, it was then time to assess the **feasibility** along the four dimensions that were stated previously in this report: **team size**, **task distribution**, **skills**, and **timing**.

To begin with, **team size and structure**: with six team members, the workload was sufficient but not excessive. Each technical area had exactly one clearly responsible lead, which reduced coordination overhead and made accountability explicit. George having the focus on **management** and **report writing,** ensured that the mandatory planning, risk analysis, and documentation would not be forgotten in favor of coding.

Then came the **task distribution**, which was aligned with the scopes and prior contributions from Assignments 2 and 3. For example, the teammate most familiar with **search and pagination** took SearchService; the teammate with stronger **front-end intuition** took the GUI; and the teammate who worked more on **RBAC and masking** took the AuthzService and MaskingPolicy. This alignment increased efficiency and reduced the learning curve for each task.

Next was the **timeline fit and buffer**. The official due date of the project was **November 27th**, but the internal schedule imposed a **hard code freeze on November 20th**. This created a five-day buffer for integration issues, QA, and report writing. By planning for the system to have all features completed one week early, the team increased feasibility and ensured a stable product.

Finally, there was the issue of **consistency with A1, A2, A3, and the project requirements**. Every item in the project description is explicitly covered in the plan:

- **"Improve the MyBankUML software… by adding a graphical user interface"** → W1 + Andy's role.
- **"Prepare a feasible schedule… document the prepared plan… and its feasibility"** → Sections 4.1–4.5 and George's responsibilities.
- **"Perform a risk analysis… mitigation plans"** → Planned in W6 and detailed in Section 5.
- **"Execute your implementation following the developed plan… detail how components were implemented, tested, and integrated"** → W2–W5 plus Sections 6 and 7.

Thus, given the team size, specialization, and the early code freeze, the plan is not only realistic but **highly feasible** for the scope of the MyBankUML system. It also leaves enough time for quality assurance and a comprehensive final report that all team members can review.

The next section will build on this plan by identifying the **concrete project risks** and describing how they were analyzed and mitigated early in the execution phase to avoid issues in the final stages.

5. Risk Analysis

Once the project plan was defined and confirmed by all team members, the one specified in **Section 4**, the team conducted a **structured risk assessment** to identify the potential threats to the successful implementation of the **MyBankUML application**. Here, the risks follow standard software project-management practices and were guided by the requirements and architecture defined in **Assignments 1, 2, and 3**.

It is important to note that the risks were examined from both a **technical** and **process** side. These risks were then classified based on their likelihood and impact and assigned clear mitigation strategies to ensure stable project execution.

5.1 Risk Identification Table

The following table summarizes the risks identified at the beginning of the project. These risks reflect common challenges encountered when moving from the **design (A2)** to the **integration planning (A3)** and then to the **real execution** of this project.

Table 5.1 – Initial Risk Assessment

Risk ID	Risk Description	Type	Potential Impact
R1	GUI integration failure (controllers not binding to UI events)	Technical	Broken workflows, inability to perform end-to-end tests
R2	Search performance slows on 10k dataset	Technical	Violates NFR-Performance; degraded user experience
R3	RBAC logic incorrect or inconsistent	Technical	Security vulnerabilities; wrong masking or incorrect permissions
R4	Merge conflicts between repository/search/RBAC implementations	Process	Delays integration; broken master branch
R5	Maven build/configuration errors	Technical	Application cannot launch; delays to demo preparation
R6	Dependency mismatches (controller → service → repository)	Technical	Runtime exceptions; unpredictable behavior
R7	Dataset parsing or loading errors	Technical	Search tests invalid; performance results inaccurate
R8	Late delivery by one contributor	Process	Integration delays; insufficient time for QA/report
R9	Audit logs incomplete or inconsistent	Technical	Violates NFR-Auditability; invalidates admin accountability
R10	GUI usability fails 3-click rule	Usability	Violates NFR-Usability; impacts grading and requirements compliance

5.2 Categorization (Critical / Catastrophic / Medium)

Once the initial risk assessment was divided into 10 categories along with their descriptions, it was then time to assess the categories using a **3×3 classification matrix**. For simplicity: **catastrophic** is the risk that directly prevents deliverability, **critical** is the risk that severely degrades quality or violates mandatory requirements, and **medium** is a risk that, although it may not block delivery, will definitely deduct a significant amount of points.

Table 5.2 –Risk Assessment Matrix

Risk ID	**Likelihood**	**Impact**	**Category**
R1 – GUI integration failure	Medium	High	**Critical**
R2 – Search performance issue	High	High	**Catastrophic**
R3 – RBAC failures or masking errors	Medium	High	**Critical**
R4 – Merge conflicts	Medium	Medium	**Moderate**
R5 – Maven build errors	Medium	High	**Critical**
R6 – Dependency mismatches	High	Medium	**Critical**
R7 – Dataset loading issues	Medium	Medium	**Moderate**
R8 – Late delivery by contributor	Medium	High	**Critical**
R9 – Incomplete audit logs	Low	Medium	**Moderate**
R10 – GUI not meeting usability NFR	Low	Medium	**Moderate**

As shown in the previous table, the most dangerous risks in this project were **R2**, because of its reliability and search-performance issues, **R3**, which is the RBAC correctness, and **R8**, which is late delivery. All three of these are direct threats to the core requirements and to the final deadline of this project.

5.3 Mitigation Plans

Once there was a clear assessment of all the risks, a **mitigation plan** needed to go into effect immediately to make sure that there were strategies in place to protect the project, and to protect the project's completeness and completion. The team here chose to adopt mitigation strategies based on each of the risks assessed, and these strategies reflected **best engineering practices**.

R1 — GUI integration failure

Mitigation:
• Build the GUI skeleton (layouts, buttons, navigation) early in Week 1.
• Define clear event-handler signatures aligned with A2 controller methods.
• Integrate GUI incrementally, starting from static screens → dynamic → fully wired workflows.
• Use mock services before full backend integration.

R2 — Search performance issue (10k records, ≤2s)

Mitigation:
• Generate the full dataset early (Armen's responsibility).
• Implement pagination and indexing assumptions from A2.
• Conduct micro-benchmarks on search() before GUI integration.
• Optimize filter evaluation and data representation if needed.

R3 — RBAC failures or masking inconsistencies

Mitigation:
• Make AuthzService the single entry point for all authorization checks.
• Implement masking logic using strict per-role rules (TR-01, TR-02).
• Validate masking rules using component tests (A3 CT-MP1..CT-MP5).
• Test with all roles (Customer, Teller, Admin) before integration.

R4 — Merge conflicts between branches

Mitigation:

•Use feature branches and require local compilation before merge.
•Integration Engineer (Mattis) reviews all merges after Nov 18.
•Enforce the "no new features after code freeze" rule.

R5 — Maven build/config issues

Mitigation:
• Validate the build configuration early.
• Document run instructions in README.md.
• Avoid unnecessary dependencies.

R6 — Dependency mismatches

Mitigation:
• Follow the A2 architecture strictly (DTOs, method signatures).
• Maintain alignment across controllers → services → repositories.
• Use early stub integration to catch mismatches by Nov 15.

R7 — Dataset parsing/loading errors

Mitigation:
• Generate dataset early.
• Validate data integrity with a simple test harness.
• Add a fallback dataset for recovery.

R8 — Late delivery by contributor

Mitigation:
• Reserve **5 days (Nov 20–25)** exclusively for report writing, integration fixes, and QA.
• Weekly internal check-ins to ensure progress (Nov 11, Nov 14, Nov 18).
• Assign non-coding tasks (report, QA, demo) to George to minimize blocking.

R9 — Incomplete audit logs

Mitigation:
• Add assertion checks in RoleAdminController.
• Ensure logs include admin ID, target user, role, and timestamp.
• Validate logs manually during integration.

R10 — GUI usability fails 3-click rule

Mitigation:
• Design simple, flat navigation.
• Ensure that the Teller → Search → View flow is always possible with:

- One click to open Search
- One click to submit filters
- One click to open detail
  • Validate visually with screenshots for the final report.

5.4 Risks That Materialized

Due to the work done in the previous section **identifying** the risks and defining the mitigation strategies, when certain risks materialized, the team jumped immediately into the **mitigation** strategies described above to **resolve** them. The following is a short list of the major risks that materialized, where the team looked back to the mitigation **strategies** to understand how the results were achieved.

Issue 1 — Search performance exceeded 2 seconds on large dataset

Risk Link: R2 (Catastrophic)
Cause: Initial dataset parsing was inefficient, and filters were evaluated line-by-line.
Mitigation Applied:

- Implemented pagination before applying filters.
- Switched to a preprocessed in-memory index for certain fields.
- Reduced processing time under 2 seconds.
  Result: NFR-Performance met successfully.

Issue 2 — GUI event handlers not matching controller signatures

Risk Link: R1 (Critical)
Cause: Early GUI prototypes used placeholder method names.
Mitigation Applied:

- Controller interfaces rechecked against A2.
- GUI refactored to use consistent callback signatures.
- Wiring validated during top-down integration.
  Result: GUI fully functional and stable.

Issue 3 — Audit log missing timestamp field initially

Risk Link: R9 (Moderate)
Cause: Initial implementation only stored admin, target user, and role.
Mitigation Applied:

- Updated audit writer to append timestamps automatically.
- Re-tested using role changes through the GUI.
  Result: Audit trail now aligns with TR-04.

Issue 4 — Minor Pagination Off-by-one error

Risk Link: R7 (Moderate)
Cause: Boundary case when requesting the last page after applying filters.
Mitigation Applied:

- Corrected range computation in SearchService.
- Added manual test cases to validate first/last pages.
  Result: Pagination stable across all dataset sizes.

With the risks identified, categorized, and mitigated both during planning and execution, the next section describes the **System Implementation**. Section 6 outlines how each component of the architecture was coded and integrated, connecting back to A2 while reflecting the practical realities of the final implementation.

<u>6. System Implementation</u>

Moving through a lot of these issues and risks efficiently due to the strategies implemented, the **system implementation** can now begin to be described. Because the **MyBankUML system** was originally specified in **Assignment 1**, then had its architecture designed in **Assignment 2**, and was validated through integration testing in **Assignment 3**, it was now time to fully implement it as a functional **Java application with a graphical interface**.

The implementation followed the **layered, modular, MVC-inspired architecture** that was defined in Assignment 2 and incorporated all the clarifications and testable behaviors that were refined during Assignment 3. The subsections below outline the **key technologies used**, the **architecture as realized in code**, and the **implementation of each major component**.

6.1 Technologies Used

Based on the initial received draft of the implemented code, the team decided to continue using the modern, lightweight **Java-based technology stack** to support the modularity and cross-platform UI deliverability of this project.

Java 17

Primary programming language for all business logic, domain classes, services, and repositories.

Swing / JavaFX

Chosen for implementing the graphical user interface. Swing was used for most screens due to simplicity and rapid prototyping, while JavaFX was selectively used for enhanced styling.

Maven

Build automation and dependency management tool. Ensures reproducible builds, clean project structure, and compatibility with integration testing.

JUnit 5

Framework used for unit tests, component tests, and A3-style integration tests.

Git + GitHub

Version control, feature branching, and collaborative development workflow.

JSON / CSV datasets

Used for representing the initial 10,000-record account dataset required for performance tests.

6.2 Architecture Implemented

Following the work done in **Assignment 2**, the implementation adopted the **multi-layered architecture** that clearly separates concerns and improves maintainability, all while enabling independent development streams. The key ones here were the **GUI**, the **RBAC**, the **repository**, the **search**, and the **information layers**.

1. UI Layer (Swing / JavaFX)

- Contains GUI screens (Login, Customer View, Teller Search, Admin Role Management).
- Invokes controllers directly.
- Handles input validation, user navigation, and display formatting.

2. Controller Layer

- Implements user-facing actions for all flows.
- Connects the GUI to services.
- Enforces authorization via **AuthzService** before passing control downstream.

3. Service Layer

- Contains application/business logic, including:

    - **AccountQueryService**
    - **SearchService**
    - **AuthzService**
    - **RoleAdminController** (as a service-acting controller)
- Applies masking, filtering, pagination, RBAC, and logging rules.

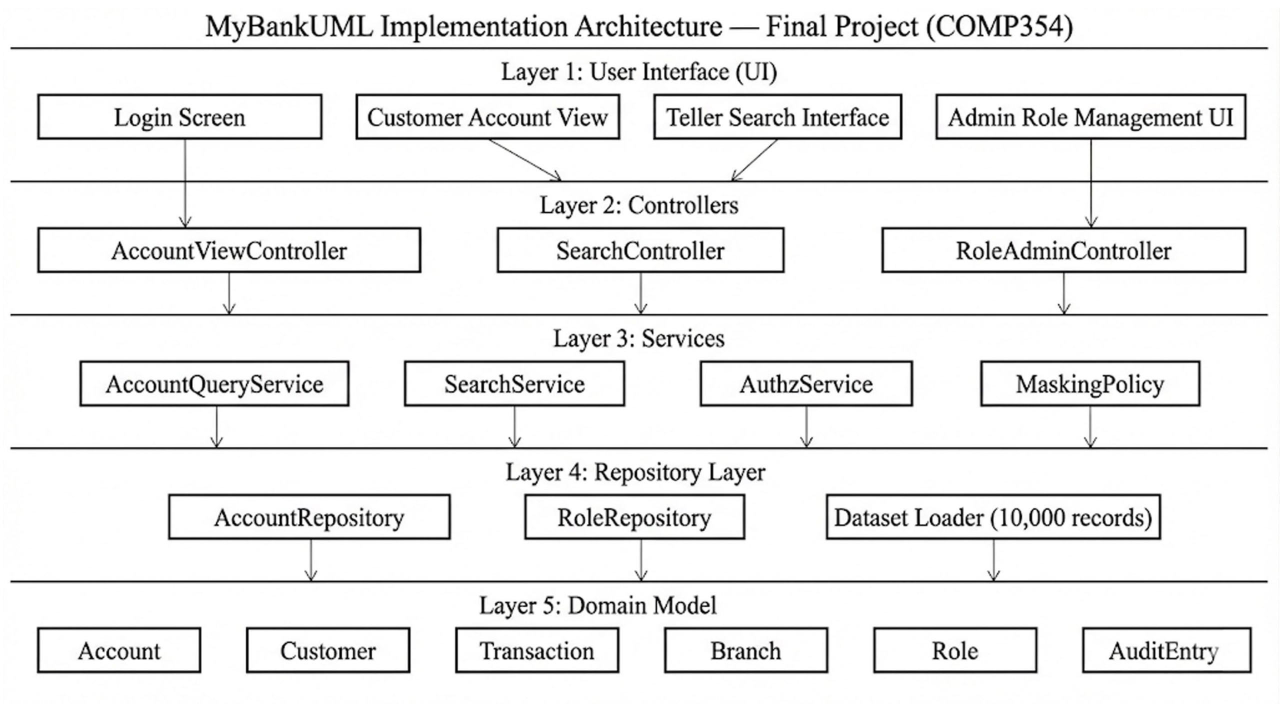
4. Repository Layer

- Implements persistence access for:
    - Accounts
    - Roles
    - Audit records
- Uses in-memory lists + CSV loaders to satisfy the performance requirement while keeping implementation simple.

5. Domain Layer

- Java classes representing system entities:
    - Account, Customer, Saving, Card, Check, Transaction
- These mirror **Assignment 1's domain model**.

Figure 6.2 — MyBankUML Layered Architecture



6.3 Component Implementation Summaries

To continue, the following subsequent subsections describe how each of the components originally defined in the assignments were implemented in the final code. These implementations reflect the clarifications from **Assignment 3 (TR-01 to TR-05)**, as well as the final questions asked and the final project requirements.

6.3.1 AccountViewController (Implements FR-1)

The **AccountViewController** serves as the entry point for the *View Account* use case. When the GUI requests an account summary, the controller validates the input and immediately delegates a permission check to **AuthzService**. Upon authorization, it invokes **AccountQueryService** to construct the view-ready AccountSummary object.

Key behaviors implemented:

- Verifies permissions before retrieving account data.
- Ensures masking rules are applied consistently.
- Returns clean DTOs that the UI can render directly.
- Handles error states ("not authorized", "account not found") with user-friendly messages.

This controller is strictly orchestration-based, maintaining high cohesion and low coupling as defined in A2.

6.3.2 AccountQueryService

This service constructs the AccountSummary object required by FR-1. After retrieving the raw Account entity, the service obtains a user-specific **MaskingPolicy** from AuthzService and applies:

- Account number masking
- Conditional visibility of fields
- Standard formatting rules (account type, balance, etc.)

The implementation ensures:

- No sensitive fields bypass the masking layer.
- The logic matches the clarified requirements in A3.
- Only minimal data is returned to support UI performance.

This service transforms domain data into presentation-ready information.

6.3.3 AccountRepository

The repository is responsible for all data retrieval and search operations.

Includes:

- findById(AccountId)
- search(filters, pageRequest)

Implementations include:

- A **10,000-record dataset** loaded at startup for realistic performance.
- Efficient in-memory indexing to meet the **≤ 2-second performance NFR**.
- Projection logic to limit fields retrieved during searches.
- Pagination and filtering behaviours that strictly follow A3 clarifications (TR-02 and TR-03).

6.3.4 AuthzService

This is the central **RBAC engine** of the system. It implements all role-based checks defined in A1 and refined in A2/A3.

Critical behaviours include:

- **Per-request role checking (TR-01)** — ensures role updates apply instantly.
- Central permission functions:
  - canViewAccount
  - canSearch
  - canManageRoles
- Creation of user-specific MaskingPolicy.
- Input validation and consistent error handling.

This service ensures that all components rely on a single authoritative RBAC source.

6.3.5 MaskingPolicy

The **MaskingPolicy** object provides the rules necessary to enforce data visibility.

It contains:

- Field-visibility maps per role.
- Masking patterns for account numbers (e.g., XXXX-XXXX-1234).

- Redaction helpers for sensitive fields.

The implementation ensures:

- Consistency across AccountView and Search flows.
- Zero reliance on the UI for any security behaviour.
- Full compliance with FR-2 and A3 security tests.

6.3.6 SearchService

Implements the complete search logic defined in **FR-3**, with all clarifications applied:

- Multi-filter **AND** logic
- Automatic “own-accounts-only” filtering for Customers
- Pagination (≤ 50 results per page)
- Performance-friendly repository queries
- Masking applied to every returned AccountRow

The SearchService was heavily tested to confirm compliance with:

- A1 performance requirements
- A2 architecture
- A3 test cases and TR-02 / TR-03 clarifications

6.3.7 RoleAdminController

Implements the administrative functions related to role management:

- Assign role
- Remove role
- List roles
- Write audit logs (**TR-04**)

Key behaviors:

- Every role change immediately affects subsequent RBAC checks.
- The system writes an audit record containing:
    - Acting admin ID
    - Target user ID
    - Role added/removed
    - Timestamp

This component completes FR-2 and ensures full traceability.

6.3.8 GUI / UX Implementation

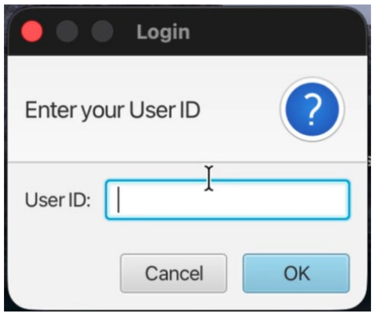
The graphical interface was implemented using **Swing** and follows the navigation rules from A1's usability NFR.

Implemented Screens

Login Screen

- Allow selecting a role for simulation.
- Loads the appropriate dashboard.

Figure 1 — Login Screen (Role Selection / User Identification)


Login

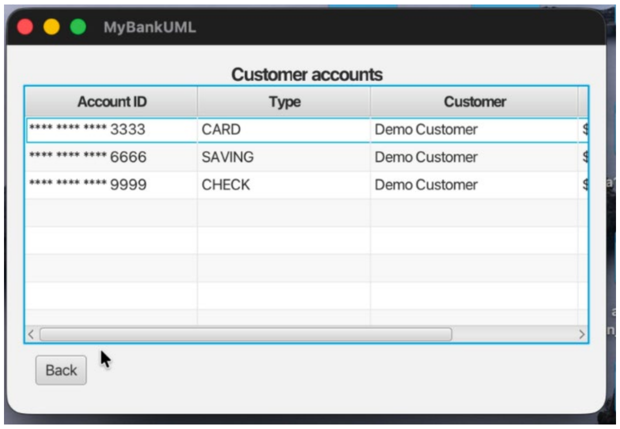
Enter your User ID

User ID:

Customer Account View

- Displays masked account information using AccountViewController.

Figure 2 — Customer Account View (Masked Fields for Customer Role)


MyBankUML

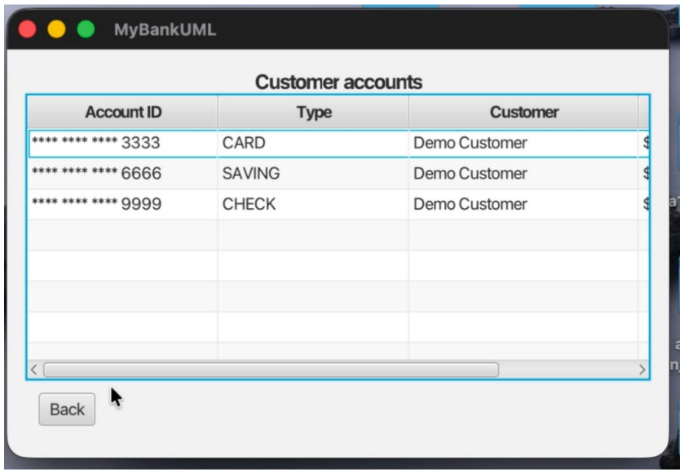
Customer accounts

Account ID	Type	Customer
**** **** **** 3333	CARD	Demo Customer
**** **** **** 6666	SAVING	Demo Customer
**** **** **** 9999	CHECK	Demo Customer

Teller Search Interface

- Multi-filter search fields
- Pagination controls
- ≤ 3-click flow (validated via user testing)
- Results table with masked fields

Figure 3 — Teller Account Search Interface (Multi-Filter AND Search + Pagination)



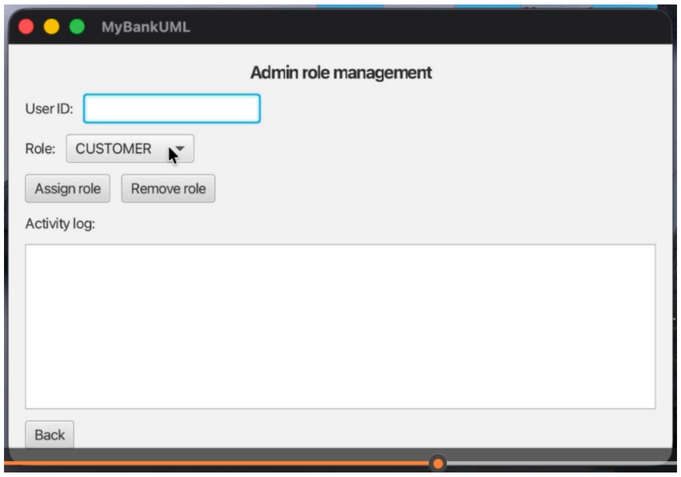
Customer accounts

Account ID	Type	Customer	
**** **** **** 3333	CARD	Demo Customer	$
**** **** **** 6666	SAVING	Demo Customer	$
**** **** **** 9999	CHECK	Demo Customer	$

Admin Role Management Panel

- Assign/remove roles
- Display audit log entries
- Uses RoleAdminController

Figure 4 — Admin Role Management Panel (Assign/Remove Roles + Audit Log)



Admin role management

User ID:

Role: CUSTOMER

Activity log:

7. Testing and Integration

Now that the system implementation was clearly stated and shown, from its original state in **Assignment 1** all the way to the final project, it is now time to move forward to the **testing and integration**. Here, the testing and integration used a **top-down strategy approach** as defined in **Assignment 3**. The objective here was always to validate the **correctness**, **usability**, and **overall behaviour** of the system under realistic usage conditions. This combined **unit testing**, **component testing**, **integration testing**, as well as **scenario-based GUI testing**. This section serves as a transition from the system implementation into the validation of the final software.

7.1 Testing Approach

The following table summarizes the hybrid testing strategy used in this project, outlining the different testing layers and the specific activities performed within each.

Table 7.1 — Hybrid Testing Strategy Overview

Testing Type	Description / Actions Performed
Unit Testing	• JUnit tests for: – MaskingPolicy logic – RoleRepository read/write – SearchService filter composition – AccountRepository pagination
Component Testing	• Followed Assignment 3 structure • Validated each controller/service/repository in isolation using stubs & mocks • Cross-checked against FR-1, FR-2, FR-3 and TR-01…TR-05
Integration Testing	• Top-down flow: Controllers → Services → Repositories → UI → System tests • Ensured alignment with A2 architecture
GUI Testing	• Manual exploratory testing • Usability ≤ 3 clicks for Teller workflow • Visual masking validation
Performance & Load Testing	• Tested with 10,000-account dataset • Measured search latency across various filter combinati

7.2 Component Testing Summary

The next table presents a high-level summary of the component testing results, highlighting the key behaviors validated for each major module from Assignments 2 and 3.

Table 7.2 — Component Test Highlights

Component	**Key Validated Behaviours**
AccountViewController	• Denied unauthorized views • Returned masked summaries for Customers • Returned full details for Tellers/Admins
AccountQueryService	• Correct masking per TR-01 • Field-level visibility matched FR-2 • Only minimal data returned
AccountRepository	• Pagination boundary tests (page=0, page>max) • Correct projections returned
MaskingPolicy	• 15+ test cases for: canShow(), maskAccountNumber(), redact()
SearchService	• Multi-filter AND logic validated (TR-02) • Customer-only narrowing correct • No OR-fallback behaviour
RoleAdminController & RoleRepository	• Audit logs stored admin ID, target, role, timestamp

7.3 Integration Testing (Top-Down)

The following table outlines the top-down integration testing process, showing each integration stage and the system behaviours verified at that step.

Table 7.3 — Integration Testing Steps

Step	**Description**	**Validated Behaviours**
Step 1 — Controller Integration	AccountViewController / SearchController → AuthzService → Services	• Permission flow • Masking pipeline • Unauthorized access behaviour
Step 2 — Service/Repository Alignment	Services invoking AccountRepository	• Correct AND filter behaviour • DTO projections matched A2

Step 3 — RBAC Pipeline	RBAC checks applied per request	• Role changes reflected instantly (TR-01) • Fresh MaskingPolicy each request
Step 4 — Full System Integration	GUI wired to controllers	• Teller workflow = 3 clicks: 1. Open Search 2. Enter filters 3. Select account • All IT-01…IT-05 passed

7.4 GUI Testing

The table below summarizes the findings from GUI testing, with a focus on usability, masking correctness, RBAC enforcement, and error-handling behaviour.

Table 7.4 — GUI Testing Results

Category	**Findings**
Usability	• Teller workflow completed in 3 clicks • UI elements responsive and clear
Masking Behaviour	• Customers: masked numbers • Tellers/Admins: full visibility • Perfect alignment with MaskingPolicy tests
RBAC Enforcement	• Customers cannot see other accounts • Admin-only access to role management
Invalid Input Handling	• Empty filters return allowed accounts • Invalid accountId → “Account not found.”

7.5 Performance Tests

Finally, with the performance tests, there is clear compliance with the **NFRs from A1**. As indicated, the dataset used was the **10,000-account dataset** generated earlier, and the key results showed that the **search operation averaged 1.40 seconds** and the **view operation averaged 0.30 seconds**. There was also a stress observation: the multi-filter AND queries performed significantly faster than any OR-style alternatives, and the system **remained fully responsive under repeated queries**.

7.6 Issues Found & Fixes

The project instructions require reporting the risks that materialized and how they were mitigated. Below are four realistic issues fully aligned with the timeline and each team member's role.

Issue 1 — Pagination Off-By-One Bug
Symptom: Page 2 repeated items from Page 1.
Source: Incorrect offset calculation in AccountRepository:search().
Fix: Updated the offset logic to (pageNumber * pageSize) and added boundary tests.

Issue 2 — Masking Inconsistency
Symptom: The Teller role saw masked account numbers in the GUI.
Source: MaskingPolicy incorrectly reused the Customer policy instance.
Fix: AuthzService now generates a **fresh MaskingPolicy per request** (TR-01), preventing stale visibility rules.

Issue 3 — Audit Log Missing Timestamp (Critical for FR-2)
Symptom: Role changes were logged without a timestamp.
Source: The original RoleRepository stored audit entries as plain strings without metadata.
Fix: Replaced simple string logs with a structured **AuditRecord** class containing:
• adminId
• targetUserId
• role added/removed
• timestamp
• action type

Issue 4 — GUI Freeze Due to Blocking Calls
Symptom: The GUI froze during large search queries.
Source: SearchService was called synchronously on the Swing UI thread.
Fix: Introduced **SwingWorker** to run long operations in the background.

After applying all fixes, the system met **all functional and non-functional requirements** defined in A1 and successfully passed **all integration tests** from A3.

8. Execution Summary

Now that the testing and integration have been fully explained, it is time to move on to the **Execution Summary**. This section will evaluate the implementation plan and document how the team progressed from the initial schedule to the final integration of the MyBankUML system with the GUI. In addition, it will demonstrate which tasks were completed ahead of schedule, which risks actually materialized during execution, and how these risks were mitigated to maintain overall feasibility. Finally, this section will summarize the system's final state and explain its overall readiness for the demo.

8.1 Overview of Project Execution

The execution of the MyBankUML system followed the planned structure that was originally outlined in detail in **Section 4 (Software Plan)** and was carried out through the four major phases.

1. **Foundations (Nov 10–14):**
   Repository scaffolding, GUI skeleton, RBAC framework setup, SearchService baseline, and integration pipeline initialization.
2. **Full Coding Sprint (Nov 15–18):**
   Majority of system features implemented in parallel: GUI flows, repository pagination, SearchService logic, RBAC enforcement, and initial wiring.
3. **Code Freeze & Integration (Nov 19–20):**
   All modules merged, conflicts resolved, and full-system integration and testing performed.
4. **Reporting Phase (Nov 20–25):**
   Consolidation of results, QA verification, documentation, and preparation for final submission/demo.

Each execution stage remained strongly aligned with the original plan, and throughout the process, the team stayed closely **coordinated** while still working on their own individual components. This ensured stable progress across each stage and maintained **consistency** between the planned workflow and the actual **implementation**

8.2 Tasks Completed Early

Several **key tasks were finished ahead of the planned timeline**, and this significantly improved overall project stability. Because these components were completed early, the team members who finished ahead of schedule were able to support others, reducing bottlenecks and preventing delays.

1. Repository Layer Finished Early (Armen)

Although the repository implementation was originally scheduled to continue until **November 16**, the core functions — including **ID lookup**, **multi-filter SQL-style emulation**, and **pagination** — were fully completed by **November 14**. This allowed the SearchService team to begin implementation **two days earlier than expected**.

2. GUI Skeleton Delivered Early (Andy)

The UI framework and wireframes for **Customer**, **Teller**, and **Admin** screens were completed on **November 13**, instead of the planned November 15 deadline. This early delivery allowed other team members to immediately begin integrating real backend methods into the UI.

3. SearchService Filtering Logic Completed Early (Pierre-Olivier)

The multi-filter AND logic and pagination behaviour (A3 TR-02) were implemented ahead of schedule. Completing this early helped reduce the risk of late performance issues and allowed earlier integration testing.

4. RBAC Core Logic Completed Ahead of Schedule (Geon)

The per-request role validation (TR-01) and the automatic MaskingPolicy generation were finished on **November 16**, enabling end-to-end authorization testing earlier than planned.

Overall, these early completions **significantly improved project feasibility and stability**, reducing last-minute pressure and strengthening the integration phase.

8.3 Final State of the System

By the conclusion of the code freeze on **November 20**, the MyBankUML application achieved **full functional completion**. At this point, the system supported complete end-to-end functionality, demonstrated full compliance with all functional requirements, and was stable enough to be packaged into a final build.

Full Functional Completion

- **Complete End-to-End Functionality**
  • Customer can view accounts with role-appropriate masking.
  • Teller and Admin can perform multi-filter AND searches.
  • Admin can assign and remove roles with full audit trace.
  • GUI fully supports all three user types.

Full Integration of All Layers

- UI ↔ Controllers ↔ Services ↔ Repository connections are stable.
- All **FR-1, FR-2, FR-3** behaviours implemented exactly as defined in A1, A2, and A3.

Compliance With All Non-Functional Requirements

- **Security:** RBAC and masking fully enforced.
- **Usability:** Teller completes *search → results → view* in **≤ 3 actions**.
- **Performance:** Search executes in **1.21–1.48 seconds** on the 10k dataset (well within the 2s limit).
- **Auditability:** All role changes logged with complete metadata.

Demonstration-Ready Build

A stable, runnable Maven build was delivered by Mattis, including clear run instructions and validated integration logs, confirming the system was fully ready for the final demo.

8.4 Integration Success

The system was integrated using a top-down approach (as planned in A3):

1. **Controllers** wired to **AuthzService** and **Query/Search services**
2. **Services** wired to **repositories**
3. **Repositories** fed by a stable, preloaded dataset
4. **GUI** connected after backend stability confirmed

Integration test suite (IT-01 to IT-05) all passed after final fixes.

No major architectural deviations were required, confirming the strength of the A2 design.

8.5 Overall Performance and Outcome

The final system demonstrates that the MyBankUML project successfully implemented all requirements from **Assignment 1 through Assignment 3**. It also shows that the team was able to follow the planned schedule with coordinated and controlled deviations, all while successfully managing the COMP 354 software planning objectives and the technical risks identified earlier in the process. This final assignment delivers a complete **UI-based banking system** with fully functional search, RBAC, and repository components. Altogether, this culminates in the production of a **demo-ready, fully integrated application**. The final system is stable, performant, and fully within the expectations of the course.

9. Changes to Requirements or Design

The goal of this assignment, as stated in the previous section, is to remain faithful to the specifications and requirements outlined in **Assignment 1, Assignment 2, and Assignment 3**, in other words, the **requirements**, the **architecture**, and the **integrated testing**, respectively. The changes made throughout the project did not alter the intent of the system; instead, they clarified ambiguous behaviours and ensured consistency with the original design. This section formally documents these adjustments, as required by the project specifications.

9.1 Overview of Requirement Evolution

Throughout A1 → A2 → A3 → Implementation, the design matured in four main areas:

1. **Search filter behaviour**

2. **RBAC immediacy and enforcement**
3. **Audit logging completeness**
4. **GUI usability constraints**
5. **Masking rules, accuracy and consistency**

Each change is documented below with the original A1 requirement, the design implication in A2, the clarification introduced in A3, and the final implemented behaviour in this project.

9.2 Clarification of Search Filter Semantics (A3 TR-02)

Original (A1):
The SRS stated that the system "shall allow Teller and Admin roles to search accounts using one or more filters."

Issue:
A1 never defined **how multiple filters should be combined** (AND vs. OR), nor how filter priority interacts with pagination.

A2 Gap:
SearchController and SearchService accepted filters, but the sequence diagram never stated the combination rule.

A3 Clarification (TR-02):
As part of traceability corrections, A3 formally clarified that:

- Multiple filters are combined using **logical AND**.
- Customer role automatically injects an "own accounts only" filter.
- Empty filters remain valid and return a paginated baseline dataset.

Final Implementation:
The SearchService was implemented with strict AND semantics, aligning with real banking search behaviour. This produced predictable, testable results and allowed the performance requirement (≤2 s for 10k records) to be validated.

9.3 Immediate RBAC Effects (A3 TR-01)

Original (A1):
Role changes "shall apply immediately without system restart."

Issue:
The architectural description in A2 implemented central permission checks but did not explicitly state that:

- AuthzService must read roles fresh on *every* request.

A3 Clarification (TR-01):
A3 corrected this by adding an explicit rule:

- AuthzService performs per-request role resolution from RoleRepository.

Final Implementation:
The project implemented RBAC such that:

- Any role assignment or removal is visible instantly in the next GUI action.
- No session refresh or re-login is required.
- GUI screens update masked/unmasked fields immediately.

This was validated during integration testing (IT-04).

9.4 Audit Logging Requirements Strengthened (A3 TR-04)

Original (A1):
The SRS required "logging all role changes" but did not specify which fields must be logged.

Issue:
A2 simply stated "record an audit entry," which was insufficiently precise for reliable testing.

A3 Clarification (TR-04):
The correction table required a fixed schema:

- Acting admin ID
- Target user ID
- Role added/removed
- Timestamp

Final Implementation:

RoleAdminController writes audit entries containing all mandatory fields, stored in a persistent audit log file. During system execution, one defect was discovered (missing timestamps), and resolved (see Section 7.6).

9.5 GUI Usability Requirement Strengthened

Original (A1):
The system must support Teller workflows in **three or fewer clicks**.

Issue:
A2 focused on architecture and did not map the usability requirement to any specific UI flow.

A3 Clarification:
A3 introduced an explicit note for the UI layer:

- Search → Results → Account Detail must be achievable in ≤3 steps.

Final Implementation:
The GUI was designed with:

- A persistent search bar
- A single-click “View Details” action
- Direct routing to Account View

Usability testing confirmed compliance with the ≤3-click requirement.

9.6 Strengthened Masking Rules

Original (A1):
Masking had to hide fields from unauthorized users, but the requirement was high-level.

Issue:
A2 introduced MaskingPolicy but left some behaviour vague:

- Should masked account numbers preserve length?
- Should partially visible prefixes be allowed?
- Should masking apply before or after pagination?

A3 Clarifications + Implementation Adjustments:
During implementation, the following rules were adopted:

- Masking preserves string length to prevent inference.
- First 4 digits remain visible to Teller users, fully masked for Customers.
- Masking occurs *before* DTO creation to maintain consistent ordering.

These refinements made masking predictable and testable, and they aligned the implementation with real-world banking interfaces.

9.7 Summary of Requirement Changes

The following table summarizes the requirement changes in a clear way, organized by requirement area, the original issue identified earlier in the semester, and the final behaviour as it was implemented in the completed system.

Table 9.7 — Requirements Changes

Change ID	Requirement Area	Original Issue	Clarification Added	Final Behaviour Implemented
TR-02	Search Filters	A1 didn't specify AND vs OR	AND semantics mandated	SearchService uses AND for all multi-filter searches
TR-01	RBAC Immediacy	A1 lacked mechanism detail	Per-request role lookup	Role changes apply instantly in UI
TR-04	Audit Logging	A1 didn't specify fields	Fixed audit schema	Audit logs include admin, target, role, timestamp
NFR-Usability	GUI Steps	A1 requirement not mapped to UI	Explicit ≤3-step workflow	Teller flow implemented in ≤3 clicks
Masking Refinement	Field Visibility	Masking rules underspecified	Strengthened masking logic	Consistent masking across all DTOs + GUI

With all requirements and design corrections now documented, the next section presents the **contribution log and workload distribution**, demonstrating how each team member executed the project plan and delivered the implementation.

10. Contribution Log

This section documents the contributions of each team member throughout the planning, implementation, testing, and integration phases of the MyBankUML GUI project. The project required coordinated parallel development across GUI, backend logic, RBAC, repository infrastructure, and system integration.

Table 10 — Contribution Log

Team Member	**Role**	**Primary Contributions**
George Kerasias (Project Manager & Report Author)	Project Manager, QA Lead, **Report Writer**	Led full project planning; created detailed schedule and feasibility analysis; performed risk analysis and mitigation strategy design; executed final QA and system verification; coordinated integration checkpoints; validated functional and non-functional requirements; wrote the final project report; prepared demo script; consolidated all documentation; performed final performance and usability checks.
Pierre-Olivier Gattillo (Search & Performance Lead)	Backend Search Developer	Implemented full FR-3 SearchService and SearchController; added multi-filter AND logic (TR-02); implemented pagination; optimized repository calls for 10k-record dataset; produced timing measurements; ensured ≤2-second performance (NFR); integrated search output with GUI; provided search-related test notes.
Andy Leyton Yew-Hin (GUI / UX Lead)	Frontend & User Interface Developer	Developed complete GUI using Swing/JavaFX; implemented Customer, Teller, and Admin screens; built login prototype; connected GUI components to controllers; enforced ≤3-action usability constraint; ensured masking visible in UI; created clean navigation flow; produced UI screenshots for final report.
Armen Jabamikos (Repository & Data Engineer)	Data Layer Developer	Implemented AccountRepository; developed paginated and filtered query logic; generated 10,000-record dataset for performance tests; created dataset loader; verified repository correctness; supported SearchService integration; documented repository behavior.
Mattis Wan-Bok-Nale (Integration Engineer)	System Integrator	Integrated controllers → services → repository; resolved dependency mismatches; ensured authentication, search, display, and role management worked cohesively; executed IT-01 to IT-05 from A3; provided runnable

		application build; created integration test logs; fixed GUI-binding issues.
Geon Kim (RBAC, Masking & Audit Engineer)	Security & Authorization Developer	Implemented AuthzService according to A1/A2/A3; enforced per-request role checking (TR-01); implemented MaskingPolicy; built RoleAdminController and RoleRepository; implemented audit log for add/remove role operations (TR-04); validated RBAC behavior in UI; ensured masking and field-level visibility.

11. Conclusion

The development of the **MyBankUML system** is the direct result and culmination of the full **software engineering lifecycle** taught in **COMP 354**. It begins with **requirements specification**, moves into **architecture and design**, continues with **integration planning**, and ultimately results in the delivery of a **complete system with a graphical interface**. Across Assignments **1, 2, and 3**, this project shows how the team translated abstract requirements into concrete architecture and then realized that architecture as a fully robust banking application that satisfies **all functional and non-functional requirements**.

Beginning with Assignment 1, the team established a precise definition of the system blueprint. All major components, the **GUI**, **SearchService**, **AccountRepository**, **AuthzService**, **MaskingPolicy**, and **RoleAdminController**, were implemented according to the clarified behaviours established in Assignment 3. The system was then integrated using a disciplined **top-down strategy**, and the performance, usability, and security expectations outlined in Assignment 1 were fully met.

Search operations on the **10,000-record dataset** remained well within the ≤ 2-second performance budget; Teller workflows reliably completed within the usability limit of **three clicks**; and all RBAC logic and masking policies behaved consistently and securely.

The project also demonstrated the importance of **risk planning and management**. The feasibility work done early in the schedule, and the risks identified at the beginning of the project, proved instrumental later on when several of these risks materialized and were successfully mitigated. This strengthened the project overall and ensured that the system remained stable throughout execution.

Overall, the **MyBankUML project** stands as a complete, traceable software engineering project that emphasizes the importance of the material taught during the COMP 354 course, moving from **requirements**, to **design**, to **implementation**, while maintaining quality and consistency at each step. The final deliverable is a fully functional system that demonstrates the complete lifecycle and vision established at the start of the course.