CAPSTONE BANKING SYSTEM DOCUMENTATION

GROUP 6

AUTHORS

KATENDE HERMAN

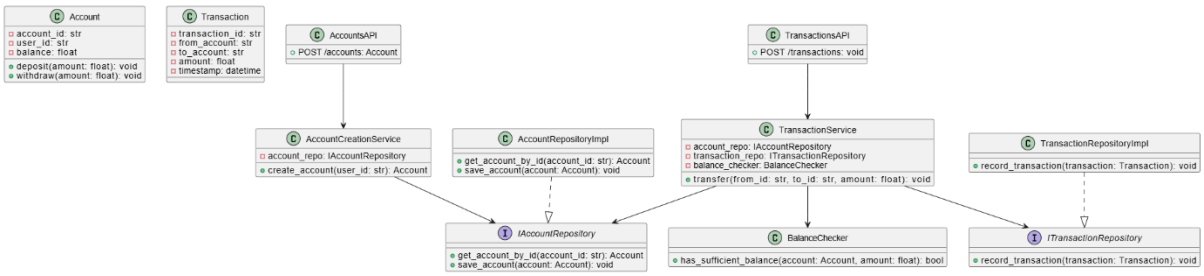
ODONGO EMMANUEL

NAKYAJA LAURA CHRISTINE

NABATANZI RANIA

UML DIAGRAM

Figure 1: This figure shows the UML class diagram for our capstone-banking system



HOW IT WORKS AND INTERGRATES.

1. Domain Layer

The **Domain Layer** is the heart of the application, containing the core business logic and rules. It defines the fundamental entities (Account, Transaction) and their behaviors, independent of external frameworks or infrastructure.

- **Entities**:
    - Account (and its subclasses **SavingsAccount**, **CheckingAccount**) encapsulates account properties (ID, type, balance) and enforces business rules (e.g., minimum balance for savings accounts, overdraft limits for checking accounts).
    - Transaction models financial actions (deposit, withdrawal) with attributes like amount, type, and timestamp.
- **Purpose**:
  This layer ensures business rules (e.g., "a savings account cannot go below $100") are centralized and reusable. It has no dependencies on external systems, making it immune to changes in databases or APIs.

2. Application Layer

The **Application Layer** orchestrates business workflows by coordinating domain objects and infrastructure components. It translates user actions (e.g., creating an account) into domain operations.

- **Services**:
    - **AccountCreationService** handles account creation, validating initial deposits and instantiating the correct Account subclass based on type (checking/savings).
    - **TransactionService** manages deposits/withdrawals, updating account balances and generating transaction records.
- **Dependencies**:
    - Relies on **Domain Layer** entities (e.g., **Account.withdraw())** to enforce rules.
    - Depends on **Infrastructure Layer** interfaces (e.g., **AccountRepository**) to persist data, adhering to the *Dependency Inversion Principle*.
- **Purpose**:
  This layer acts as a mediator, ensuring use cases (e.g., "transfer funds") are executed consistently while keeping domain logic decoupled from technical details.

3. Infrastructure Layer

The **Infrastructure Layer** provides implementations for external interactions, such as data storage or third-party services. It adapts the core system to real-world tools.

- **Components**:
    - InMemoryAccountRepository and InMemoryTransactionRepository implement repository interfaces, using Python dictionaries to simulate databases.
    - **Adapters**: In a real-world scenario, these could be replaced with SQL/NoSQL databases or external APIs without altering the Domain/Application layers.
- **Integration**:
    - Implements interfaces defined in the **Application Layer** (e.g., **AccountRepository**), allowing the Application Layer to remain agnostic to storage details.
    - For example, when **TransactionService** saves a transaction, it calls **transaction_repo.save_transaction(),** which delegates to the in-memory repository.
- **Purpose**:
  This layer handles I/O operations, ensuring the core logic remains pure and testable.

4. Presentation Layer

The **Presentation Layer** exposes the system's functionality to external users or systems, typically via RESTful APIs.

- **Components**:
    - FastAPI routes (e.g., POST /accounts, POST /deposit) define endpoints that map HTTP requests to application services.
    - Uses **Pydantic** models (e.g., **CreateAccountRequest**) to validate input and serialize responses.
- **Integration**:
    - Routes inject dependencies (e.g., **AccountCreationService**) from the **Application Layer**, which in turn rely on **Infrastructure Layer** repositories.
    - For example, a POST /accounts request triggers **AccountCreationService.create_account(),** which uses the repository to persist the new account.
- **Purpose**:
  This layer acts as the system's "front door," translating HTTP requests into domain actions and returning structured responses (e.g., JSON).

Layer Integration Flow

1. **Request Handling**:
    - A user sends an HTTP request (e.g., POST /accounts with account details).
    - The **Presentation Layer** (FastAPI) validates the request and forwards it to the **Application Layer** (**AccountCreationService**).
2. **Business Logic Execution**:
    - The **Application Layer** uses **Domain Layer** entities (e.g., **SavingsAccount**) to enforce rules (e.g., minimum deposit).
    - It delegates data persistence to the **Infrastructure Layer** (e.g., **InMemoryAccountRepository.create_account**()).
3. **Response Generation**:
    - Results are serialized by the **Presentation Layer** into JSON and returned to the user.