ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA

Rapport de Projet

Gestion d'un Système Bancaire

Réalisé par : WAEL BOUHAYA
MOHAMED REDA HABCHI
LAYLA IBEN KHALDOUNE

Encadré par : Prof. HAIN MUSTAPHA

Module : POO Python

Filière : IAGI-1

Année Universitaire 2025-2026

Table des matières

Introduction

Ce rapport présente l'analyse technique et architecturale d'un système de gestion bancaire développé en Python. Le projet implémente une application complète de gestion de comptes bancaires avec interface graphique, base de données relationnelle et logique métier conforme aux règles bancaires.

1 Architecture Globale

1.1 Composants Principaux

Le système repose sur une architecture à trois niveaux :

Composant	**Technologie**	**Fonction**
Couche Présentation	Tkinter	Interface graphique utilisateur
Couche Métier	Python (POO)	Logique applicative et règles métier
Couche Données	SQLite	Persistance et gestion des données

TABLE 1 – Architecture à trois niveaux du système

1.2 Organisation du Code

Le projet se compose de trois fichiers principaux :

- `pro_bank_app.py` : Application principale contenant l'interface graphique et la logique métier
- `populate_db.py` : Script d'initialisation de la base de données avec jeu de données de test
- `bank.db` : Fichier de base de données SQLite

L'application principale est structurée selon les principes de la Programmation Orientée Objet (POO), avec une classe dédiée pour chaque écran (Login, Interface Admin, Interface Client).

2 Couche Données

2.1 Gestionnaire de Base de Données

La classe `DatabaseManager` constitue le point d'accès unique à la base de données. Cette approche centralise toutes les opérations SQL et garantit une séparation claire entre la logique de présentation et la logique de données.

Extrait du code - Initialisation du DatabaseManager :

```
class DatabaseManager:
    def __init__(self, db_name="bank.db"):
        self.db_name = db_name
        self.conn = sqlite3.connect(self.db_name)
        self.cur = self.conn.cursor()
        self.create_tables()

    def create_tables(self):
        self.cur.execute("""
            CREATE TABLE IF NOT EXISTS customers (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                email TEXT UNIQUE NOT NULL,
                phone TEXT
            )
        """)
        # Cr ation des autres tables : accounts, users,
           transactions
        self.conn.commit()
```

Listing 1 – Classe DatabaseManager

2.2 Modèle de Données Relationnel

La base de données est structurée autour de quatre tables interconnectées :

Table customers
Stocke les informations personnelles des clients (nom, email, téléphone).

Table users Gère l'authentification avec les identifiants de connexion, mots de passe et rôles (admin ou customer).

Table accounts
Contient les comptes bancaires avec leur numéro, solde et type, liés à un client via `customer_id`.

Table transactions
Enregistre l'historique complet des opérations (montant, type, date) pour chaque compte.

2.3 Script d'Initialisation

Le fichier `populate_db.py` automatise la création d'un environnement de test fonctionnel :

```
if __name__ == "__main__":
    DB_NAME = "bank.db"
    try:
        conn = sqlite3.connect(DB_NAME)
        cur = conn.cursor()

        clear_data(conn)
        customer_ids = create_customers(cur, n=50)
        create_users_for_customers(cur, customer_ids)
        account_ids = create_accounts(cur, customer_ids)
        create_transactions(cur, account_ids)

        conn.commit()
```

Listing 2 – Séquence d'initialisation

La bibliothèque `Faker` est utilisée pour générer des données réalistes de manière automatisée, facilitant ainsi les phases de test et de démonstration.

3 Système d'Authentification et de Contrôle d'Accès

3.1 Mécanisme d'Authentification

Le système implémente un processus d'authentification basé sur la vérification des identifiants en base de données.

```
def check_login(self, username, password):
    self.cur.execute("""
        SELECT role, customer_id
        FROM users
        WHERE username = ? AND password = ?
    """, (username, password))

    result = self.cur.fetchone()
    if result:
        return result[0], result[1]
    return None, None
```

Listing 3 – Vérification des identifiants

Mesure de sécurité : Les requêtes SQL utilisent des paramètres préparés (`?`) pour prévenir les attaques par injection SQL.

3.2 Contrôle d'Accès Basé sur les Rôles

Le système applique un contrôle d'accès différencié selon le rôle de l'utilisateur :

```
def on_login_success(self, role, customer_id):
    if self.current_frame:
        self.current_frame.destroy()

    if role == 'admin':
        self.current_frame = AdminInterface(
            self, self.db, self.show_login_screen
        )
    else:
        self.current_frame = CustomerInterface(
            self, self.db, customer_id, self.show_login_screen
        )
```

Listing 4 – Redirection selon le rôle

— **Rôle Admin :** Accès complet à toutes les données et fonctionnalités de gestion
— **Rôle Customer :** Accès limité aux comptes personnels via `customer_id`

4 Logique Métier des Opérations Bancaires

4.1 Opération de Retrait

L'opération de retrait implémente une règle métier fondamentale : la vérification obligatoire de la disponibilité des fonds.

```
def withdraw(self, account_id, amount):
    current_balance = self.get_account_balance(account_id)
    if current_balance < amount:
        return "Insufficient funds."
    new_balance = current_balance - amount
    self.cur.execute(
        "UPDATE accounts SET balance = ? WHERE id = ?",
        (new_balance, account_id)
    )
    self.cur.execute("""
        INSERT INTO transactions (account_id, type, amount,
           date)
        VALUES (?, 'withdraw', ?, ?)
    """, (account_id, amount, date.today()))
    self.conn.commit()
    return "Success"
```

Listing 5 – Méthode de retrait

Le processus garantit qu'aucun retrait ne peut être effectué si le solde est insuffisant, empêchant ainsi les découverts non autorisés.

4.2 Opération de Dépôt

L'opération de dépôt suit une logique similaire mais sans contrainte de solde minimum :

1. Mise à jour du solde du compte
2. Enregistrement de la transaction dans l'historique
3. Validation de l'opération via `commit()`

4.3 Opération de Virement

Le virement constitue l'opération la plus complexe, combinant un retrait et un dépôt dans une transaction atomique.

```
def transfer(self, from_account_number, to_account_number,
    amount):
     from_id, from_balance = self.get_account_id_and_balance(
         from_account_number
     )
     to_id, to_balance = self.get_account_id_and_balance(
         to_account_number
     )

     if from_id is None or to_id is None:
         return "Account not found."

     if from_id == to_id:
         return "Cannot transfer to the same account."

     if from_balance < amount:
         return "Insufficient funds."

     self._update_balance(from_id, -amount)
     self._update_balance(to_id, amount)
     self._record_transaction(from_id, 'transfer_out', amount)
     self._record_transaction(to_id, 'transfer_in', amount)

     self.conn.commit()
     return "Success"
```

Listing 6 – Méthode de virement

Principe d'atomicité : Les opérations de débit et de crédit sont exécutées dans une même transaction. Si une erreur survient avant le `commit()`, aucune modification n'est enregistrée, garantissant ainsi la cohérence des données.

5 Interface Administrateur

5.1 Consultation des Transactions

L'interface administrateur permet une supervision globale des transactions via des requêtes SQL avancées utilisant des jointures.

```
def get_all_transactions(self):
    self.cur.execute("""
        SELECT t.date, c.name, a.account_number, t.type, t.
           amount
        FROM transactions t
        JOIN accounts a ON t.account_id = a.id
        JOIN customers c ON a.customer_id = c.id
        ORDER BY t.date DESC
    """)
    return self.cur.fetchall()
```

Listing 7 – Requête avec jointures multiples

Les jointures SQL permettent d'agréger les données de plusieurs tables pour produire des rapports complets et intelligibles, associant par exemple les transactions aux noms des clients propriétaires des comptes.

5.2 Opérations CRUD

L'administrateur dispose de fonctionnalités complètes de gestion (Create, Read, Update, Delete) :

— **Création :** Ajout de nouveaux clients (`INSERT INTO customers`)

— **Lecture :** Consultation des données clients et comptes

— **Modification :** Mise à jour des informations (`UPDATE customers SET...`)

— **Suppression :** Fermeture de comptes (`DELETE FROM accounts`)

Ces opérations sont encapsulées dans la classe `DatabaseManager` et exposées via l'interface `AdminInterface`.

6 Analyse Technique et Évaluation

6.1 Points Forts

Architecture modulaire
La séparation claire entre interface, logique métier et données facilite la maintenance et l'évolution du système.

Respect des principes POO
L'utilisation de classes dédiées améliore la lisibilité et la réutilisabilité du code.

Sécurité des requêtes
L'utilisation systématique de requêtes paramétrées prévient les vulnérabilités liées aux injections SQL.

Intégrité des données
L'implémentation de l'atomicité des transactions garantit la cohérence de la base de données.

Règles métier conformes
Les vérifications de solde et les contraintes bancaires sont correctement appliquées.

6.2 Technologies Utilisées

— **Python 3.x :** Langage principal pour la logique applicative
— **Tkinter :** Framework GUI natif pour l'interface graphique
— **SQLite3 :** Système de base de données relationnelle léger et performant
— **Faker :** Bibliothèque de génération de données de test

6.3 Conformité aux Standards

Le projet démontre une maîtrise des concepts suivants :
2
— Programmation orientée objet
— Gestion de bases de données relationnelles
— Requêtes SQL avancées (jointures, agrégations)
— Développement d'interfaces graphiques
— Gestion de transactions ACID

Conclusion

Ce système de gestion bancaire constitue une application fonctionnelle et complète, démontrant la capacité à concevoir et implémenter une solution logicielle intégrant interface utilisateur, logique métier et persistance des données. L'architecture choisie respecte les bonnes pratiques de développement et garantit la fiabilité des opérations bancaires critiques.

Le projet illustre une compréhension approfondie des concepts de développement d'applications de gestion et de manipulation de bases de données relationnelles.