I'll provide the step-by-step code for each file in the credit card statement generator project. I'll assume we're implementing this for Maybank (Malaysia) as an example and use Python with relevant libraries for the implementation.

Let's start with creating each file in the structure:

**1. database/init.py**

# Empty \_\_init\_\_.py file to make the directory a package

**2. database/connection.py**

import pyodbc

import logging

from utils.config import get\_db\_config

from utils.error\_handler import DatabaseError

class DatabaseConnection:

"""Class to handle database connections and operations"""

def \_\_init\_\_(self):

"""Initialize database connection"""

self.connection = None

self.cursor = None

self.config = get\_db\_config()

def connect(self):

"""Establish connection to the database"""

try:

connection\_string = (

f"DRIVER={{SQL Server}};"

f"SERVER={self.config['server']};"

f"DATABASE={self.config['database']};"

f"UID={self.config['username']};"

f"PWD={self.config['password']};"

)

self.connection = pyodbc.connect(connection\_string)

self.cursor = self.connection.cursor()

logging.info("Database connection established successfully")

return True

except Exception as e:

logging.error(f"Failed to connect to database: {str(e)}")

raise DatabaseError(f"Database connection error: {str(e)}")

def disconnect(self):

"""Close the database connection"""

if self.connection:

self.connection.close()

logging.info("Database connection closed")

def execute\_query(self, query, params=None):

"""Execute a query and return results"""

try:

if params:

self.cursor.execute(query, params)

else:

self.cursor.execute(query)

# If the query is a SELECT statement, return results

if query.strip().upper().startswith('SELECT'):

columns = [column[0] for column in self.cursor.description]

return [dict(zip(columns, row)) for row in self.cursor.fetchall()]

else:

self.connection.commit()

return True

except Exception as e:

self.connection.rollback()

logging.error(f"Query execution error: {str(e)}")

raise DatabaseError(f"Query execution error: {str(e)}")

def get\_customer\_data(self, customer\_id):

"""Get customer information"""

query = """

SELECT c.\*, a.account\_number, a.credit\_limit, a.available\_credit,

a.current\_balance, a.statement\_date, a.payment\_due\_date,

a.minimum\_payment, a.previous\_balance, a.language\_preference

FROM customers c

JOIN accounts a ON c.customer\_id = a.customer\_id

WHERE c.customer\_id = ?

"""

return self.execute\_query(query, (customer\_id,))

def get\_transactions(self, account\_number, start\_date, end\_date):

"""Get transactions for a specific period"""

query = """

SELECT t.transaction\_id, t.transaction\_date, t.posting\_date,

t.description, t.amount, t.transaction\_type, t.mcc\_code,

t.reference\_number

FROM transactions t

WHERE t.account\_number = ?

AND t.posting\_date BETWEEN ? AND ?

ORDER BY t.posting\_date DESC, t.transaction\_id DESC

"""

return self.execute\_query(query, (account\_number, start\_date, end\_date))

def get\_rewards(self, account\_number):

"""Get rewards information for an account"""

query = """

SELECT r.rewards\_id, r.points\_earned, r.points\_redeemed,

r.points\_balance, r.points\_expiring, r.expiry\_date

FROM rewards r

WHERE r.account\_number = ?

"""

return self.execute\_query(query, (account\_number,))

**3. database/schema.sql**

-- Database Schema for Maybank Credit Card Statement Generator

-- Create Customers Table

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

first\_name NVARCHAR(50) NOT NULL,

last\_name NVARCHAR(50) NOT NULL,

address\_line1 NVARCHAR(100) NOT NULL,

address\_line2 NVARCHAR(100),

city NVARCHAR(50) NOT NULL,

state NVARCHAR(50) NOT NULL,

postal\_code NVARCHAR(10) NOT NULL,

country NVARCHAR(50) NOT NULL DEFAULT 'Malaysia',

email NVARCHAR(100),

phone NVARCHAR(20),

membership\_since DATE NOT NULL,

customer\_type NVARCHAR(20) DEFAULT 'Regular'

);

-- Create Accounts Table

CREATE TABLE accounts (

account\_id INT PRIMARY KEY,

account\_number NVARCHAR(20) NOT NULL UNIQUE,

customer\_id INT NOT NULL,

card\_type NVARCHAR(50) NOT NULL,

credit\_limit DECIMAL(15, 2) NOT NULL,

available\_credit DECIMAL(15, 2) NOT NULL,

current\_balance DECIMAL(15, 2) NOT NULL,

previous\_balance DECIMAL(15, 2) NOT NULL,

statement\_date DATE NOT NULL,

payment\_due\_date DATE NOT NULL,

minimum\_payment DECIMAL(15, 2) NOT NULL,

annual\_fee DECIMAL(15, 2) DEFAULT 0.00,

interest\_rate DECIMAL(5, 2) NOT NULL,

language\_preference NVARCHAR(10) DEFAULT 'EN', -- EN for English, MS for Malay

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

-- Create Transactions Table

CREATE TABLE transactions (

transaction\_id INT PRIMARY KEY,

account\_number NVARCHAR(20) NOT NULL,

transaction\_date DATE NOT NULL,

posting\_date DATE NOT NULL,

description NVARCHAR(200) NOT NULL,

amount DECIMAL(15, 2) NOT NULL,

transaction\_type NVARCHAR(20) NOT NULL, -- 'Purchase', 'Payment', 'Fee', 'Interest', etc.

mcc\_code NVARCHAR(10), -- Merchant Category Code

reference\_number NVARCHAR(50),

FOREIGN KEY (account\_number) REFERENCES accounts(account\_number)

);

-- Create Rewards Table

CREATE TABLE rewards (

rewards\_id INT PRIMARY KEY,

account\_number NVARCHAR(20) NOT NULL,

points\_earned INT NOT NULL DEFAULT 0,

points\_redeemed INT NOT NULL DEFAULT 0,

points\_balance INT NOT NULL DEFAULT 0,

points\_expiring INT DEFAULT 0,

expiry\_date DATE,

FOREIGN KEY (account\_number) REFERENCES accounts(account\_number)

);

-- Create Statements Table

CREATE TABLE statements (

statement\_id INT PRIMARY KEY,

account\_number NVARCHAR(20) NOT NULL,

statement\_date DATE NOT NULL,

statement\_period\_start DATE NOT NULL,

statement\_period\_end DATE NOT NULL,

opening\_balance DECIMAL(15, 2) NOT NULL,

closing\_balance DECIMAL(15, 2) NOT NULL,

total\_payments DECIMAL(15, 2) NOT NULL DEFAULT 0.00,

total\_purchases DECIMAL(15, 2) NOT NULL DEFAULT 0.00,

total\_fees DECIMAL(15, 2) NOT NULL DEFAULT 0.00,

total\_interest DECIMAL(15, 2) NOT NULL DEFAULT 0.00,

minimum\_payment DECIMAL(15, 2) NOT NULL,

payment\_due\_date DATE NOT NULL,

pdf\_generated BIT DEFAULT 0,

generation\_date DATETIME,

FOREIGN KEY (account\_number) REFERENCES accounts(account\_number)

);

-- Create Indexes for better performance

CREATE INDEX idx\_transactions\_account\_date ON transactions(account\_number, posting\_date);

CREATE INDEX idx\_customers\_name ON customers(last\_name, first\_name);

CREATE INDEX idx\_statements\_account\_date ON statements(account\_number, statement\_date);

**4. database/test\_data.sql**

-- Insert test data into Customers table

INSERT INTO customers (customer\_id, first\_name, last\_name, address\_line1, address\_line2, city, state, postal\_code, country, email, phone, membership\_since, customer\_type)

VALUES

(1001, 'Ahmad', 'Abdullah', '123 Jalan Merdeka', 'Taman Sejahtera', 'Kuala Lumpur', 'Wilayah Persekutuan', '50100', 'Malaysia', 'ahmad.abdullah@email.com', '+60123456789', '2015-03-15', 'Platinum'),

(1002, 'Siti', 'Rahman', '45 Jalan Bukit Bintang', 'Unit 12-3', 'Kuala Lumpur', 'Wilayah Persekutuan', '55100', 'Malaysia', 'siti.rahman@email.com', '+60198765432', '2018-06-22', 'Gold'),

(1003, 'Raj', 'Kumar', '78 Jalan Ampang', NULL, 'Kuala Lumpur', 'Wilayah Persekutuan', '50450', 'Malaysia', 'raj.kumar@email.com', '+60173456789', '2017-11-10', 'Regular');

-- Insert test data into Accounts table

INSERT INTO accounts (account\_id, account\_number, customer\_id, card\_type, credit\_limit, available\_credit, current\_balance, previous\_balance, statement\_date, payment\_due\_date, minimum\_payment, annual\_fee, interest\_rate, language\_preference)

VALUES

(101, '5526-8942-3567-1234', 1001, 'Maybank 2 Cards Premier', 50000.00, 32456.75, 17543.25, 15890.50, '2025-03-15', '2025-04-05', 877.16, 600.00, 15.00, 'EN'),

(102, '5526-8942-7865-5678', 1002, 'Maybank Islamic Ikhwan Card-i Gold', 20000.00, 18750.25, 1249.75, 2500.00, '2025-03-15', '2025-04-05', 62.49, 300.00, 17.50, 'MS'),

(103, '5526-8942-1122-9012', 1003, 'Maybank Manchester United Visa Card', 10000.00, 9800.00, 200.00, 0.00, '2025-03-15', '2025-04-05', 10.00, 150.00, 18.00, 'EN');

-- Insert test data into Transactions table for account 1

INSERT INTO transactions (transaction\_id, account\_number, transaction\_date, posting\_date, description, amount, transaction\_type, mcc\_code, reference\_number)

VALUES

(10001, '5526-8942-3567-1234', '2025-02-18', '2025-02-20', 'PAYMENT - THANK YOU', -5000.00, 'Payment', NULL, 'P987654321'),

(10002, '5526-8942-3567-1234', '2025-02-22', '2025-02-23', 'PARKSON KL SURIA KLCC', 1250.75, 'Purchase', '5311', 'T123456789'),

(10003, '5526-8942-3567-1234', '2025-02-25', '2025-02-27', 'LAZADA MALAYSIA SDN BHD', 549.90, 'Purchase', '5942', 'T123456790'),

(10004, '5526-8942-3567-1234', '2025-03-01', '2025-03-03', 'SHELL JALAN AMPANG', 120.00, 'Purchase', '5542', 'T123456791'),

(10005, '5526-8942-3567-1234', '2025-03-05', '2025-03-06', 'NETFLIX.COM', 45.90, 'Purchase', '4816', 'T123456792'),

(10006, '5526-8942-3567-1234', '2025-03-07', '2025-03-08', 'GRAB TRANSPORT', 25.00, 'Purchase', '4121', 'T123456793'),

(10007, '5526-8942-3567-1234', '2025-03-10', '2025-03-11', 'ANNUAL FEE', 600.00, 'Fee', NULL, 'F123456789'),

(10008, '5526-8942-3567-1234', '2025-03-12', '2025-03-12', 'FINANCE CHARGE', 237.50, 'Interest', NULL, 'I123456789');

-- Insert test data into Transactions table for account 2

INSERT INTO transactions (transaction\_id, account\_number, transaction\_date, posting\_date, description, amount, transaction\_type, mcc\_code, reference\_number)

VALUES

(20001, '5526-8942-7865-5678', '2025-02-18', '2025-02-20', 'PAYMENT - THANK YOU', -1500.00, 'Payment', NULL, 'P287654321'),

(20002, '5526-8942-7865-5678', '2025-02-24', '2025-02-25', 'MYDIN SUBANG JAYA', 350.25, 'Purchase', '5411', 'T223456789'),

(20003, '5526-8942-7865-5678', '2025-03-01', '2025-03-02', 'WATSONS MID VALLEY', 125.50, 'Purchase', '5912', 'T223456790'),

(20004, '5526-8942-7865-5678', '2025-03-08', '2025-03-09', 'GARDENIA BAKERIES KL', 24.00, 'Purchase', '5462', 'T223456791');

-- Insert test data into Transactions table for account 3

INSERT INTO transactions (transaction\_id, account\_number, transaction\_date, posting\_date, description, amount, transaction\_type, mcc\_code, reference\_number)

VALUES

(30001, '5526-8942-1122-9012', '2025-03-05', '2025-03-06', 'MANCHESTER UNITED STORE', 200.00, 'Purchase', '5655', 'T323456789');

-- Insert test data into Rewards table

INSERT INTO rewards (rewards\_id, account\_number, points\_earned, points\_redeemed, points\_balance, points\_expiring, expiry\_date)

VALUES

(1, '5526-8942-3567-1234', 35000, 10000, 25000, 5000, '2025-12-31'),

(2, '5526-8942-7865-5678', 5000, 2000, 3000, 1000, '2025-12-31'),

(3, '5526-8942-1122-9012', 400, 0, 400, 0, NULL);

-- Insert test data into Statements table

INSERT INTO statements (statement\_id, account\_number, statement\_date, statement\_period\_start, statement\_period\_end, opening\_balance, closing\_balance, total\_payments, total\_purchases, total\_fees, total\_interest, minimum\_payment, payment\_due\_date)

VALUES

(1, '5526-8942-3567-1234', '2025-03-15', '2025-02-16', '2025-03-15', 15890.50, 17543.25, 5000.00, 5815.25, 600.00, 237.50, 877.16, '2025-04-05'),

(2, '5526-8942-7865-5678', '2025-03-15', '2025-02-16', '2025-03-15', 2500.00, 1249.75, 1500.00, 249.75, 0.00, 0.00, 62.49, '2025-04-05'),

(3, '5526-8942-1122-9012', '2025-03-15', '2025-02-16', '2025-03-15', 0.00, 200.00, 0.00, 200.00, 0.00, 0.00, 10.00, '2025-04-05');

**5. models/init.py**

# Empty \_\_init\_\_.py file to make the directory a package

**6. models/customer.py**

class Customer:

"""Customer model representing customer data"""

def \_\_init\_\_(self, customer\_data):

"""Initialize customer with data from database"""

self.customer\_id = customer\_data.get('customer\_id')

self.first\_name = customer\_data.get('first\_name')

self.last\_name = customer\_data.get('last\_name')

self.address\_line1 = customer\_data.get('address\_line1')

self.address\_line2 = customer\_data.get('address\_line2')

self.city = customer\_data.get('city')

self.state = customer\_data.get('state')

self.postal\_code = customer\_data.get('postal\_code')

self.country = customer\_data.get('country')

self.email = customer\_data.get('email')

self.phone = customer\_data.get('phone')

self.membership\_since = customer\_data.get('membership\_since')

self.customer\_type = customer\_data.get('customer\_type')

def get\_full\_name(self):

"""Return the customer's full name"""

return f"{self.first\_name} {self.last\_name}"

def get\_full\_address(self):

"""Return the customer's full address as a formatted string"""

address\_parts = [self.address\_line1]

if self.address\_line2:

address\_parts.append(self.address\_line2)

address\_parts.append(f"{self.city}, {self.state} {self.postal\_code}")

address\_parts.append(self.country)

return "\n".join(address\_parts)

def to\_dict(self):

"""Convert customer data to dictionary"""

return {

'customer\_id': self.customer\_id,

'full\_name': self.get\_full\_name(),

'address': self.get\_full\_address(),

'email': self.email,

'phone': self.phone,

'membership\_since': self.membership\_since,

'customer\_type': self.customer\_type

}

**7. models/account.py**

from datetime import datetime

class Account:

"""Account model representing credit card account data"""

def \_\_init\_\_(self, account\_data):

"""Initialize account with data from database"""

self.account\_id = account\_data.get('account\_id')

self.account\_number = account\_data.get('account\_number')

self.customer\_id = account\_data.get('customer\_id')

self.card\_type = account\_data.get('card\_type')

self.credit\_limit = account\_data.get('credit\_limit')

self.available\_credit = account\_data.get('available\_credit')

self.current\_balance = account\_data.get('current\_balance')

self.previous\_balance = account\_data.get('previous\_balance')

self.statement\_date = account\_data.get('statement\_date')

self.payment\_due\_date = account\_data.get('payment\_due\_date')

self.minimum\_payment = account\_data.get('minimum\_payment')

self.annual\_fee = account\_data.get('annual\_fee')

self.interest\_rate = account\_data.get('interest\_rate')

self.language\_preference = account\_data.get('language\_preference')

def get\_masked\_account\_number(self):

"""Return the masked account number for display"""

if not self.account\_number:

return ""

parts = self.account\_number.split('-')

if len(parts) == 4:

# Mask middle parts

masked\_parts = [parts[0], 'XXXX', 'XXXX', parts[3]]

return '-'.join(masked\_parts)

else:

# If format is not as expected, mask everything except last 4 digits

raw\_number = self.account\_number.replace('-', '')

return 'XXXX-XXXX-XXXX-' + raw\_number[-4:]

def get\_statement\_period(self, end\_date=None):

"""Calculate statement period based on statement date"""

if end\_date is None:

end\_date = self.statement\_date

# If end\_date is a string, convert to datetime

if isinstance(end\_date, str):

end\_date = datetime.strptime(end\_date, '%Y-%m-%d').date()

# Calculate start date (usually one month before end date)

if end\_date.month == 1:

start\_month = 12

start\_year = end\_date.year - 1

else:

start\_month = end\_date.month - 1

start\_year = end\_date.year

# Adjust for different month lengths

start\_day = min(end\_date.day, 28) # Safe for February

# Create start date

start\_date = datetime(start\_year, start\_month, start\_day).date()

return {

'start\_date': start\_date,

'end\_date': end\_date

}

def to\_dict(self):

"""Convert account data to dictionary"""

return {

'account\_number': self.account\_number,

'masked\_account\_number': self.get\_masked\_account\_number(),

'card\_type': self.card\_type,

'credit\_limit': self.credit\_limit,

'available\_credit': self.available\_credit,

'current\_balance': self.current\_balance,

'previous\_balance': self.previous\_balance,

'statement\_date': self.statement\_date,

'payment\_due\_date': self.payment\_due\_date,

'minimum\_payment': self.minimum\_payment,

'annual\_fee': self.annual\_fee,

'interest\_rate': self.interest\_rate,

'language\_preference': self.language\_preference

}

**8. models/transaction.py**

class Transaction:

"""Transaction model representing credit card transaction data"""

def \_\_init\_\_(self, transaction\_data):

"""Initialize transaction with data from database"""

self.transaction\_id = transaction\_data.get('transaction\_id')

self.account\_number = transaction\_data.get('account\_number')

self.transaction\_date = transaction\_data.get('transaction\_date')

self.posting\_date = transaction\_data.get('posting\_date')

self.description = transaction\_data.get('description')

self.amount = transaction\_data.get('amount')

self.transaction\_type = transaction\_data.get('transaction\_type')

self.mcc\_code = transaction\_data.get('mcc\_code')

self.reference\_number = transaction\_data.get('reference\_number')

def is\_debit(self):

"""Check if transaction is a debit (charge)"""

return self.amount > 0

def is\_credit(self):

"""Check if transaction is a credit (payment)"""

return self.amount < 0

def get\_formatted\_amount(self, currency="RM"):

"""Return formatted amount with currency symbol"""

if self.is\_debit():

return f"{currency} {abs(self.amount):.2f}"

else:

return f"{currency} {abs(self.amount):.2f} CR"

def get\_category(self):

"""Get transaction category based on type and MCC code"""

if self.transaction\_type == 'Payment':

return 'Payment'

elif self.transaction\_type == 'Fee':

return 'Fee'

elif self.transaction\_type == 'Interest':

return 'Interest'

elif self.transaction\_type == 'Purchase':

# Categorize based on MCC code

if self.mcc\_code in ['5411', '5422', '5462']:

return 'Groceries'

elif self.mcc\_code in ['5812', '5813', '5814']:

return 'Dining'

elif self.mcc\_code in ['4121', '4111', '4112']:

return 'Transportation'

elif self.mcc\_code in ['5311', '5651', '5691']:

return 'Shopping'

else:

return 'Other Purchases'

else:

return 'Other'

def to\_dict(self):

"""Convert transaction data to dictionary"""

return {

'transaction\_id': self.transaction\_id,

'transaction\_date': self.transaction\_date,

'posting\_date': self.posting\_date,

'description': self.description,

'amount': self.amount,

'formatted\_amount': self.get\_formatted\_amount(),

'transaction\_type': self.transaction\_type,

'category': self.get\_category(),

'reference\_number': self.reference\_number

}

**9. processors/init.py**

# Empty \_\_init\_\_.py file to make the directory a package

**10. processors/financial\_calculator.py**

class FinancialCalculator:

"""Calculates financial data for credit card statement"""

@staticmethod

def calculate\_minimum\_payment(current\_balance, minimum\_payment\_percentage=5.0, minimum\_amount=10.0):

"""

Calculate minimum payment based on current balance

Args:

current\_balance: The current outstanding balance

minimum\_payment\_percentage: The percentage for minimum payment (default 5%)

minimum\_amount: The minimum amount regardless of percentage (default RM10)

Returns:

The calculated minimum payment amount

"""

calculated\_amount = current\_balance \* (minimum\_payment\_percentage / 100)

return max(calculated\_amount, minimum\_amount) if current\_balance > 0 else 0

@staticmethod

def calculate\_interest(previous\_balance, payments, interest\_rate, days\_in\_month=30):

"""

Calculate interest charges based on average daily balance

Args:

previous\_balance: Previous statement balance

payments: List of payment transactions with amounts and dates

interest\_rate: Annual interest rate percentage

days\_in\_month: Number of days in the billing cycle

Returns:

The calculated interest amount

"""

# Simple implementation using average daily balance method

# In a real application, this would be more complex with exact day calculations

# Calculate average daily balance

total\_balance = previous\_balance \* days\_in\_month

# Subtract payments based on when they were made in the cycle

for payment in payments:

# Get day of payment in the cycle (assume payment.posting\_date is a datetime object)

if hasattr(payment, 'posting\_date') and payment.posting\_date:

days\_remaining = days\_in\_month - payment.posting\_date.day

total\_balance -= abs(payment.amount) \* days\_remaining

average\_daily\_balance = total\_balance / days\_in\_month

# Calculate monthly interest

monthly\_rate = interest\_rate / 12 / 100

interest = average\_daily\_balance \* monthly\_rate

return round(interest, 2)

@staticmethod

def calculate\_statement\_summary(transactions):

"""

Calculate statement summary from transactions

Args:

transactions: List of transaction objects

Returns:

Dictionary with summary information

"""

summary = {

'total\_debits': 0,

'total\_credits': 0,

'total\_fees': 0,

'total\_interest': 0,

'total\_purchases': 0,

'transactions\_count': len(transactions)

}

for transaction in transactions:

amount = abs(transaction.amount)

if transaction.transaction\_type == 'Fee':

summary['total\_fees'] += amount

summary['total\_debits'] += amount

elif transaction.transaction\_type == 'Interest':

summary['total\_interest'] += amount

summary['total\_debits'] += amount

elif transaction.transaction\_type == 'Purchase':

summary['total\_purchases'] += amount

summary['total\_debits'] += amount

elif transaction.transaction\_type == 'Payment':

summary['total\_credits'] += amount

return summary

**11. processors/data\_processor.py**

from models.customer import Customer

from models.account import Account

from models.transaction import Transaction

from processors.financial\_calculator import FinancialCalculator

from utils.error\_handler import DataProcessingError

import logging

class DataProcessor:

"""Processes data for credit card statement generation"""

def \_\_init\_\_(self, db\_connection):

"""

Initialize data processor with database connection

Args:

db\_connection: Database connection object

"""

self.db = db\_connection

self.calculator = FinancialCalculator()

def process\_statement\_data(self, customer\_id, statement\_date=None):

"""

Process all data needed for statement generation

Args:

customer\_id: ID of the customer

statement\_date: Optional statement date (defaults to most recent)

Returns:

Dictionary with all processed data for statement generation

"""

try:

# Get customer and account data

customer\_data = self.db.get\_customer\_data(customer\_id)

if not customer\_data:

raise DataProcessingError(f"No data found for customer ID: {customer\_id}")

# Create models from database data

customer = Customer(customer\_data[0])

account = Account(customer\_data[0])

# Get statement period

statement\_period = account.get\_statement\_period(statement\_date)

# Get transactions for the period

transactions\_data = self.db.get\_transactions(

account.account\_number,

statement\_period['start\_date'],

statement\_period['end\_date']

)

transactions = [Transaction(t) for t in transactions\_data]

# Get rewards data

rewards\_data = self.db.get\_rewards(account.account\_number)

rewards = rewards\_data[0] if rewards\_data else {}

# Calculate statement summary

summary = self.calculator.calculate\_statement\_summary(transactions)

# Categorize transactions

categorized\_transactions = self.\_categorize\_transactions(transactions)

# Prepare final statement data

statement\_data = {

'customer': customer.to\_dict(),

'account': account.to\_dict(),

'statement\_period': statement\_period,

'transactions': [t.to\_dict() for t in transactions],

'categorized\_transactions': categorized\_transactions,

'summary': summary,

'rewards': rewards,

'language': account.language\_preference

}

return statement\_data

except Exception as e:

logging.error(f"Error processing statement data: {str(e)}")

raise DataProcessingError(f"Failed to process statement data: {str(e)}")

def \_categorize\_transactions(self, transactions):

"""

Categorize transactions by type for easier processing in templates

Args:

transactions: List of transaction objects

Returns:

Dictionary with transactions organized by category

"""

categories = {

'payments': [],

'purchases': [],

'fees': [],

'interest': [],

'other': []

}

Let me continue with the remaining files for the credit card statement generator project:

## 11. processors/data\_processor.py (continued)

def \_categorize\_transactions(self, transactions):

"""

Categorize transactions by type for easier processing in templates

Args:

transactions: List of transaction objects

Returns:

Dictionary with transactions organized by category

"""

categories = {

'payments': [],

'purchases': [],

'fees': [],

'interest': [],

'other': []

}

for transaction in transactions:

if transaction.transaction\_type == 'Payment':

categories['payments'].append(transaction)

elif transaction.transaction\_type == 'Purchase':

categories['purchases'].append(transaction)

elif transaction.transaction\_type == 'Fee':

categories['fees'].append(transaction)

elif transaction.transaction\_type == 'Interest':

categories['interest'].append(transaction)

else:

categories['other'].append(transaction)

return categories

## 12. pdf\_generator/****init****.py

# Empty \_\_init\_\_.py file to make the directory a package

## 13. pdf\_generator/template\_manager.py

import os

import logging

from utils.error\_handler import TemplateError

from utils.language import get\_translation

class TemplateManager:

"""Manages templates for different banks and languages"""

def \_\_init\_\_(self, bank\_name, language='EN'):

"""

Initialize template manager for specific bank and language

Args:

bank\_name: Name of the bank (e.g., 'maybank')

language: Language code (default 'EN' for English)

"""

self.bank\_name = bank\_name.lower()

self.language = language.upper()

self.translations = get\_translation(self.language)

# Load bank-specific template module

try:

template\_module = \_\_import\_\_(f'pdf\_generator.templates.{self.bank\_name}',

fromlist=['get\_template\_data'])

self.template\_data = template\_module.get\_template\_data(self.language)

except ImportError:

logging.error(f"Template for bank '{bank\_name}' not found")

raise TemplateError(f"Template for bank '{bank\_name}' not found")

except Exception as e:

logging.error(f"Error loading template: {str(e)}")

raise TemplateError(f"Error loading template: {str(e)}")

def get\_template\_path(self):

"""Get path to HTML template file"""

template\_dir = os.path.join(os.path.dirname(\_\_file\_\_), 'templates', self.bank\_name)

template\_file = f"statement\_template\_{self.language.lower()}.html"

template\_path = os.path.join(template\_dir, template\_file)

if not os.path.exists(template\_path):

logging.error(f"Template file not found: {template\_path}")

raise TemplateError(f"Template file not found: {template\_path}")

return template\_path

def get\_template\_assets(self):

"""Get dictionary of template assets (images, logos, etc.)"""

asset\_dir = os.path.join(os.path.dirname(\_\_file\_\_), 'templates', self.bank\_name, 'assets')

assets = {}

# Load all assets from directory

if os.path.exists(asset\_dir):

for filename in os.listdir(asset\_dir):

file\_path = os.path.join(asset\_dir, filename)

if os.path.isfile(file\_path):

assets[filename] = file\_path

return assets

def get\_template\_data(self):

"""Get template data including text and formatting options"""

return self.template\_data

def translate(self, key):

"""Translate a key to the current language"""

return self.translations.get(key, key)

## 14. pdf\_generator/pdf\_builder.py

import os

import logging

from datetime import datetime

import jinja2

import pdfkit

from pdf\_generator.template\_manager import TemplateManager

from utils.error\_handler import PDFGenerationError

class PDFBuilder:

"""Builds PDF statements based on statement data and templates"""

def \_\_init\_\_(self, bank\_name, output\_dir='output'):

"""

Initialize PDF builder

Args:

bank\_name: Name of the bank

output\_dir: Directory to save output PDF files

"""

self.bank\_name = bank\_name.lower()

self.output\_dir = output\_dir

# Create output directory if it doesn't exist

if not os.path.exists(output\_dir):

os.makedirs(output\_dir)

def generate\_pdf(self, statement\_data):

"""

Generate PDF statement from statement data

Args:

statement\_data: Processed statement data

Returns:

Path to the generated PDF file

"""

try:

# Get language from statement data

language = statement\_data.get('language', 'EN')

# Initialize template manager

template\_manager = TemplateManager(self.bank\_name, language)

template\_path = template\_manager.get\_template\_path()

template\_assets = template\_manager.get\_template\_assets()

# Prepare data for template

template\_data = {

'statement': statement\_data,

'template': template\_manager.get\_template\_data(),

'translate': template\_manager.translate,

'generation\_date': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),

'assets': template\_assets

}

# Load template with Jinja2

template\_loader = jinja2.FileSystemLoader(os.path.dirname(template\_path))

template\_env = jinja2.Environment(loader=template\_loader)

template = template\_env.get\_template(os.path.basename(template\_path))

# Render HTML

html\_content = template.render(\*\*template\_data)

# Create a temporary HTML file

temp\_html\_path = os.path.join(self.output\_dir, 'temp\_statement.html')

with open(temp\_html\_path, 'w', encoding='utf-8') as html\_file:

html\_file.write(html\_content)

# Generate PDF filename

customer\_id = statement\_data['customer']['customer\_id']

account\_number = statement\_data['account']['masked\_account\_number'].replace('-', '').replace('X', '')

statement\_date = statement\_data['statement\_period']['end\_date'].strftime('%Y%m%d')

pdf\_filename = f"{self.bank\_name}\_statement\_{customer\_id}\_{account\_number}\_{statement\_date}.pdf"

pdf\_path = os.path.join(self.output\_dir, pdf\_filename)

# Convert HTML to PDF using pdfkit (wkhtmltopdf wrapper)

options = {

'page-size': 'A4',

'margin-top': '10mm',

'margin-right': '10mm',

'margin-bottom': '10mm',

'margin-left': '10mm',

'encoding': 'UTF-8',

'no-outline': None,

'enable-local-file-access': None

}

pdfkit.from\_file(temp\_html\_path, pdf\_path, options=options)

# Clean up temporary HTML file

os.remove(temp\_html\_path)

logging.info(f"PDF statement generated: {pdf\_path}")

return pdf\_path

except Exception as e:

logging.error(f"Error generating PDF: {str(e)}")

raise PDFGenerationError(f"Failed to generate PDF statement: {str(e)}")

## 15. pdf\_generator/templates/maybank.py

def get\_template\_data(language='EN'):

"""

Get template data for Maybank credit card statements

Args:

language: Language code ('EN' for English, 'MS' for Malay)

Returns:

Dictionary with template configuration

"""

template = {

'bank\_name': 'Maybank',

'logo\_path': 'maybank\_logo.png',

'primary\_color': '#FFC800', # Maybank yellow

'secondary\_color': '#000000', # Black

'font\_family': 'Arial, Helvetica, sans-serif',

'header\_font\_family': 'Arial, Helvetica, sans-serif',

'footer\_text': 'Maybank Berhad (Registration No. 196001000142)',

'contact\_info': {

'EN': {

'address': 'Maybank Card Centre\nMenara Maybank, 100 Jalan Tun Perak\n50050 Kuala Lumpur',

'phone': 'Customer Service Hotline: 1300-88-6688',

'email': 'service@maybank.com.my',

'website': 'www.maybank.com.my'

},

'MS': {

'address': 'Pusat Kad Maybank\nMenara Maybank, 100 Jalan Tun Perak\n50050 Kuala Lumpur',

'phone': 'Talian Khidmat Pelanggan: 1300-88-6688',

'email': 'service@maybank.com.my',

'website': 'www.maybank.com.my'

}

},

'sections': [

{

'name': 'header',

'order': 1,

'title': {

'EN': 'CREDIT CARD STATEMENT',

'MS': 'PENYATA KAD KREDIT'

}

},

{

'name': 'summary',

'order': 2,

'title': {

'EN': 'STATEMENT SUMMARY',

'MS': 'RINGKASAN PENYATA'

}

},

{

'name': 'account\_details',

'order': 3,

'title': {

'EN': 'ACCOUNT DETAILS',

'MS': 'BUTIRAN AKAUN'

}

},

{

'name': 'transactions',

'order': 4,

'title': {

'EN': 'TRANSACTION DETAILS',

'MS': 'BUTIRAN TRANSAKSI'

}

},

{

'name': 'rewards',

'order': 5,

'title': {

'EN': 'REWARDS SUMMARY',

'MS': 'RINGKASAN GANJARAN'

}

},

{

'name': 'payment\_instructions',

'order': 6,

'title': {

'EN': 'PAYMENT INFORMATION',

'MS': 'MAKLUMAT PEMBAYARAN'

}

},

{

'name': 'footer',

'order': 7,

'title': {

'EN': 'IMPORTANT INFORMATION',

'MS': 'MAKLUMAT PENTING'

}

}

],

'transaction\_headers': {

'EN': {

'date': 'Date',

'description': 'Description',

'amount': 'Amount (RM)'

},

'MS': {

'date': 'Tarikh',

'description': 'Keterangan',

'amount': 'Amaun (RM)'

}

},

'payment\_instructions': {

'EN': [

'Pay online through Maybank2u',

'Cash/cheque payment at any Maybank branch',

'Payment via ATM',

'JomPAY Online (Biller Code: 50643)'

],

'MS': [

'Bayar dalam talian melalui Maybank2u',

'Bayaran tunai/cek di mana-mana cawangan Maybank',

'Pembayaran melalui ATM',

'JomPAY Dalam Talian (Kod Pengebil: 50643)'

]

}

}

return template

## 16. Create Maybank HTML Template

This would be stored at pdf\_generator/templates/maybank/statement\_template\_en.html:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Maybank Credit Card Statement</title>

<style>

body {

font-family: Arial, Helvetica, sans-serif;

font-size: 10pt;

color: #000000;

margin: 0;

padding: 0;

}

.header {

background-color: #FFC800;

padding: 15px;

overflow: hidden;

}

.logo {

float: left;

width: 150px;

}

.header-title {

float: right;

font-size: 18pt;

font-weight: bold;

margin-top: 20px;

}

.statement-info {

padding: 15px;

background-color: #F5F5F5;

border-bottom: 1px solid #CCCCCC;

}

.customer-info {

float: left;

width: 60%;

}

.date-info {

float: right;

width: 35%;

text-align: right;

}

.summary-box {

clear: both;

margin: 15px;

border: 1px solid #CCCCCC;

padding: 10px;

}

.summary-title {

font-size: 12pt;

font-weight: bold;

border-bottom: 2px solid #FFC800;

padding-bottom: 5px;

margin-bottom: 10px;

}

.summary-table {

width: 100%;

border-collapse: collapse;

}

.summary-table td {

padding: 5px;

}

.summary-table .label {

font-weight: bold;

width: 60%;

}

.highlight {

background-color: #FFFBE6;

}

.transaction-table {

width: 100%;

border-collapse: collapse;

margin-top: 10px;

}

.transaction-table th {

background-color: #EFEFEF;

padding: 5px;

text-align: left;

border-bottom: 1px solid #CCCCCC;

}

.transaction-table td {

padding: 5px;

border-bottom: 1px solid #EFEFEF;

}

.amount {

text-align: right;

}

.payment {

color: #006600;

}

.section {

margin: 15px;

page-break-inside: avoid;

}

.section-title {

font-size: 12pt;

font-weight: bold;

border-bottom: 2px solid #FFC800;

padding-bottom: 5px;

margin-bottom: 10px;

}

.footer {

padding: 15px;

background-color: #F5F5F5;

font-size: 8pt;

text-align: center;

border-top: 1px solid #CCCCCC;

margin-top: 20px;

}

.clearfix {

clear: both;

}

</style>

</head>

<body>

<!-- Header -->

<div class="header">

<img class="logo" src="{{ assets['maybank\_logo.png'] }}" alt="Maybank Logo">

<div class="header-title">{{ translate('CREDIT\_CARD\_STATEMENT') }}</div>

<div class="clearfix"></div>

</div>

<!-- Statement Info -->

<div class="statement-info">

<div class="customer-info">

<strong>{{ statement.customer.full\_name }}</strong><br>

{{ statement.customer.address|replace('\n', '<br>')|safe }}

</div>

<div class="date-info">

<strong>{{ translate('STATEMENT\_DATE') }}:</strong> {{ statement.statement\_period.end\_date|strftime('%d %b %Y') }}<br>

<strong>{{ translate('PAYMENT\_DUE\_DATE') }}:</strong> {{ statement.account.payment\_due\_date|strftime('%d %b %Y') }}<br>

<strong>{{ translate('ACCOUNT\_NUMBER') }}:</strong> {{ statement.account.masked\_account\_number }}

</div>

<div class="clearfix"></div>

</div>

<!-- Summary Box -->

<div class="summary-box">

<div class="summary-title">{{ translate('STATEMENT\_SUMMARY') }}</div>

<table class="summary-table">

<tr class="highlight">

<td class="label">{{ translate('NEW\_BALANCE') }}</td>

<td class="amount">RM {{ statement.account.current\_balance|format\_currency }}</td>

</tr>

<tr class="highlight">

<td class="label">{{ translate('MINIMUM\_PAYMENT') }}</td>

<td class="amount">RM {{ statement.account.minimum\_payment|format\_currency }}</td>

</tr>

<tr class="highlight">

<td class="label">{{ translate('PAYMENT\_DUE\_DATE') }}</td>

<td>{{ statement.account.payment\_due\_date|strftime('%d %b %Y') }}</td>

</tr>

<tr>

<td class="label">{{ translate('PREVIOUS\_BALANCE') }}</td>

<td class="amount">RM {{ statement.account.previous\_balance|format\_currency }}</td>

</tr>

<tr>

<td class="label">{{ translate('TOTAL\_PAYMENTS') }}</td>

<td class="amount payment">RM {{ statement.summary.total\_credits|format\_currency }}</td>

</tr>

<tr>

<td class="label">{{ translate('TOTAL\_PURCHASES\_AND\_CHARGES') }}</td>

<td class="amount">RM {{ statement.summary.total\_debits|format\_currency }}</td>

</tr>

</table>

</div>

<!-- Account Details -->

<div class="section">

<div class="section-title">{{ translate('ACCOUNT\_DETAILS') }}</div>

<table class="summary-table">

<tr>

<td class="label">{{ translate('CARD\_TYPE') }}</td>

<td>{{ statement.account.card\_type }}</td>

</tr>

<tr>

<td class="label">{{ translate('CREDIT\_LIMIT') }}</td>

<td class="amount">RM {{ statement.account.credit\_limit|format\_currency }}</td>

</tr>

<tr>

<td class="label">{{ translate('AVAILABLE\_CREDIT') }}</td>

<td class="amount">RM {{ statement.account.available\_credit|format\_currency }}</td>

</tr>

<tr>

<td class="label">{{ translate('STATEMENT\_PERIOD') }}</td>

<td>{{ statement.statement\_period.start\_date|strftime('%d %b %Y') }} - {{ statement.statement\_period.end\_date|strftime('%d %b %Y') }}</td>

</tr>

</table>

</div>

<!-- Transaction Details -->

<div class="section">

<div class="section-title">{{ translate('TRANSACTION\_DETAILS') }}</div>

{% if statement.categorized\_transactions.payments %}

<div style="margin-top: 15px;">

<strong>{{ translate('PAYMENTS') }}</strong>

<table class="transaction-table">

<tr>

<th>{{ template.transaction\_headers[statement.language].date }}</th>

<th>{{ template.transaction\_headers[statement.language].description }}</th>

<th class="amount">{{ template.transaction\_headers[statement.language].amount }}</th>

</tr>

{% for transaction in statement.categorized\_transactions.payments %}

<tr>

<td>{{ transaction.posting\_date|strftime('%d/%m/%Y') }}</td>

<td>{{ transaction.description }}</td>

<td class="amount payment">{{ transaction.formatted\_amount }}</td>

</tr>

{% endfor %}

</table>

</div>

{% endif %}

{% if statement.categorized\_transactions.purchases %}

<div style="margin-top: 15px;">

<strong>{{ translate('PURCHASES\_AND\_CHARGES') }}</strong>

<table class="transaction-table">

<tr>

<th>{{ template.transaction\_headers[statement.language].date }}</th>

<th>{{ template.transaction\_headers[statement.language].description }}</th>

<th class="amount">{{ template.transaction\_headers[statement.language].amount }}</th>

</tr>

{% for transaction in statement.categorized\_transactions.purchases %}

<tr>

<td>{{ transaction.posting\_date|strftime('%d/%m/%Y') }}</td>

<td>{{ transaction.description }}</td>

<td class="amount">{{ transaction.formatted\_amount }}</td>

</tr>

{% endfor %}

</table>

</div>

{% endif %}

{% if statement.categorized\_transactions.fees %}

<div style="margin-top: 15px;">

<strong>{{ translate('FEES') }}</strong>

<table class="transaction-table">

<tr>

<th>{{ template.transaction\_headers[statement.language].date }}</th>

<th>{{ template.transaction\_headers[statement.language].description }}</th>

<th class="amount">{{ template.transaction\_headers[statement.language].amount }}</th>

</tr>

{% for transaction in statement.categorized\_transactions.fees %}

<tr>

<td>{{ transaction.posting\_date|strftime('%d/%m/%Y') }}</td>

<td>{{ transaction.description }}</td>

<td class="amount">{{ transaction.formatted\_amount }}</td>

</tr>

{% endfor %}

</table>

</div>

{% endif %}

{% if statement.categorized\_transactions.interest %}

<div style="margin-top: 15px;">

<strong>{{ translate('INTEREST\_CHARGES') }}</strong>

<table class="transaction-table">

<tr>

<th>{{ template.transaction\_headers[statement.language].date }}</th>

<th>{{ template.transaction\_headers[statement.language].description }}</th>

<th class="amount">{{ template.transaction\_headers[statement.language].amount }}</th>

</tr>

{% for transaction in statement.categorized\_transactions.interest %}

<tr>

<td>{{ transaction.posting\_date|strftime('%d/%m/%Y') }}</td>

<td>{{ transaction.description }}</td>

<td class="amount">{{ transaction.formatted\_amount }}</td>

</tr>

{% endfor %}

</table>

</div>

{% endif %}

</div>

<!-- Rewards Summary -->

<div class="section">

<div class="section-title">{{ translate('REWARDS\_SUMMARY') }}</div>

<table class="summary-table">

<tr>

<td class="label">{{ translate('POINTS\_BALANCE') }}</td>

<td>{{ statement.rewards.points\_balance|default(0) }}</td>

</tr>

<tr>

<td class="label">{{ translate('POINTS\_EARNED\_THIS\_PERIOD') }}</td>

<td>{{ statement.rewards.points\_earned|default(0) }}</td>

</tr>

<tr>

<td class="label">{{ translate('POINTS\_REDEEMED') }}</td>

<td>{{ statement.rewards.points\_redeemed|default(0) }}</td>

</tr>

{% if statement.rewards.points\_expiring and statement.rewards.expiry\_date %}

<tr>

<td class="label">{{ translate('POINTS\_EXPIRING') }}</td>

<td>{{ statement.rewards.points\_expiring }} ({{ translate('EXPIRES\_ON') }} {{ statement.rewards.expiry\_date|strftime('%d %b %Y') }})</td>

</tr>

{% endif %}

</table>

</div>

<!-- Payment Instructions -->

<div class="section">

<div class="section-title">{{ translate('PAYMENT\_INFORMATION') }}</div>

<div>

<strong>{{ translate('PAYMENT\_METHODS') }}:</strong>

<ul>

{% for instruction in template.payment\_instructions[statement.language] %}

<li>{{ instruction }}</li>

{% endfor %}

</ul>

</div>

</div>

<!-- Footer -->

<div class="footer">

{{ template.footer\_text }}<br>

{{ template.contact\_info[statement.language].address|replace('\n', ', ')|safe }} | {{ template.contact\_info[statement.language].phone }} | {{ template.contact\_info[statement.language].email }}<br>

{{ template.contact\_info[statement.language].website }}

</div>

</body>

</html>

## 17. utils/****init****.py

# Empty \_\_init\_\_.py file to make the directory a package

## 18. utils/config.py

import os

import json

import logging

class ConfigurationError(Exception):

"""Exception raised for configuration errors"""

pass

def \_load\_config\_file():

"""Load configuration from JSON file"""

config\_path = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'config.json')

# Check if config file exists

if not os.path.exists(config\_path):

# Create default config

default\_config = {

'database': {

'server': 'localhost',

'database': 'credit\_card\_statements',

'username': 'statement\_user',

'password': 'your\_password',

'connection\_timeout': 30

},

'pdf': {

'output\_directory': 'output',

'page\_size': 'A4',

'dpi': 300

},

'logging': {

'level': 'INFO',

'file': 'app.log'

}

}

# Write default config

with open(config\_path, 'w') as config\_file:

json.dump(default\_config, config\_file, indent=4)

logging.warning(f"Config file not found. Created default config at {config\_path}")

return default\_config

# Load existing config

try:

with open(config\_path, 'r') as config\_file:

return json.load(config\_file)

except Exception as e:

logging.error(f"Error loading config file: {str(e)}")

raise ConfigurationError(f"Failed to load configuration: {str(e)}")

# Load configuration once at module import

\_config = \_load\_config\_file()

def get\_config():

"""Get full configuration"""

return \_config

def get\_db\_config():

"""Get database configuration"""

return \_config.get('database', {})

def get\_pdf\_config():

"""Get PDF generation configuration"""

return \_config.get('pdf', {})

def get\_logging\_config():

"""Get logging configuration"""

return \_config.get('logging', {})

## 19. utils/error\_handler.py

class StatementGeneratorError(Exception):

"""Base exception class for statement generator errors"""

pass

class DatabaseError(StatementGeneratorError):

"""Exception raised for database errors"""

pass

class DataProcessingError(StatementGeneratorError):

"""Exception raised for data processing errors"""

pass

class TemplateError(StatementGeneratorError):

"""Exception raised for template errors"""

pass

class PDFGenerationError(StatementGeneratorError):

"""Exception raised for PDF generation errors"""

pass

class ValidationError(StatementGeneratorError):

"""Exception raised for validation errors"""

pass

def handle\_error(error, logger=None):

"""

Handle and log errors

Args:

error: The error exception

logger: Optional logger object

Returns:

Dictionary with error information

"""

error\_info = {

'type': error.\_\_class\_\_.\_\_name\_\_,

'message': str(error),

'handled': True

}

if logger:

logger.error(f"{error\_info['type']}: {error\_info['message']}")

return error\_info

## 20. utils/language.py

def get\_translation(language\_code='EN'):

"""

Get translations for the specified language

Args:

language\_code: Language code ('EN' for English, 'MS' for Malay)

Returns:

Dictionary with translated strings

"""

translations = {

'EN': {

'CREDIT\_CARD\_STATEMENT': 'CREDIT CARD STATEMENT',

'STATEMENT\_DATE': 'Statement Date',

'PAYMENT\_DUE\_DATE': 'Payment Due Date',

'ACCOUNT\_NUMBER': 'Account Number',

'STATEMENT\_SUMMARY': 'STATEMENT SUMMARY',

'NEW\_BALANCE': 'New Balance',

'MINIMUM\_PAYMENT': 'Minimum Payment',

'PREVIOUS\_BALANCE': 'Previous Balance',

'TOTAL\_PAYMENTS': 'Total Payments',

'TOTAL\_PURCHASES\_AND\_CHARGES': 'Total Purchases and Charges',

'ACCOUNT\_DETAILS': 'ACCOUNT DETAILS',

'CARD\_TYPE': 'Card Type',

'CREDIT\_LIMIT': 'Credit Limit',

'AVAILABLE\_CREDIT': 'Available Credit',

'STATEMENT\_PERIOD': 'Statement Period',

'TRANSACTION\_DETAILS': 'TRANSACTION DETAILS',

'PAYMENTS': 'PAYMENTS',

'PURCHASES\_AND\_CHARGES': 'PURCHASES AND CHARGES',

'FEES': 'FEES',

'INTEREST\_CHARGES': 'INTEREST CHARGES',

'REWARDS\_SUMMARY': 'REWARDS SUMMARY',

'POINTS\_BALANCE': 'Points Balance',

'POINTS\_EARNED\_THIS\_PERIOD': 'Points Earned This Period',

'POINTS\_REDEEMED': 'Points Redeemed',

'POINTS\_EXPIRING': 'Points Expiring',

'EXPIRES\_ON': 'Expires on',

'PAYMENT\_INFORMATION': 'PAYMENT INFORMATION',

'PAYMENT\_METHODS': 'Payment Methods',

'IMPORTANT\_INFORMATION': 'IMPORTANT INFORMATION'

},

'MS': {

'CREDIT\_CARD\_STATEMENT': 'PENYATA KAD KREDIT',

'STATEMENT\_DATE': 'Tarikh Penyata',

'PAYMENT\_DUE\_DATE': 'Tarikh Pembayaran',

'ACCOUNT\_NUMBER': 'Nombor Akaun',

'STATEMENT\_SUMMARY': 'RINGKASAN PENYATA',

'NEW\_BALANCE': 'Baki Baru'

}}

20.test/test\_calculations.py

# tests/test\_calculations.py

import unittest

import os

import sys

from datetime import datetime, date, timedelta

from decimal import Decimal

# Add project root to path

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

from processors.financial\_calculator import FinancialCalculator

from models.customer import Customer

from models.account import Account

from models.transaction import Transaction

from database.connection import db

class TestFinancialCalculations(unittest.TestCase):

    """Test financial calculations for credit card statements"""

    def setUp(self):

        """Set up test environment"""

        self.connection = db.connect()

        # Clear test data

        cursor = self.connection.cursor()

        cursor.execute("SET FOREIGN\_KEY\_CHECKS = 0")

        cursor.execute("TRUNCATE TABLE transactions")

        cursor.execute("TRUNCATE TABLE accounts")

        cursor.execute("TRUNCATE TABLE customers")

        cursor.execute("TRUNCATE TABLE billing\_cycles")

        cursor.execute("TRUNCATE TABLE interest\_calculations")

        cursor.execute("SET FOREIGN\_KEY\_CHECKS = 1")

        self.connection.commit()

        # Create test customer and account

        self.test\_customer = Customer(

            first\_name="Test",

            last\_name="User",

            email="test@example.com",

            phone="+60123456789",

            address\_line1="123 Test Street",

            city="Test City",

            state="Test State",

            postal\_code="12345",

            country="Malaysia",

            date\_of\_birth=date(1990, 1, 1)

        )

        self.test\_customer.save()

        self.test\_account = Account(

            customer\_id=self.test\_customer.customer\_id,

            account\_number="TEST0001234567",

            card\_number="5196061234567890",

            card\_type="MASTERCARD",

            credit\_limit=10000.00,

            current\_balance=0.00,

            statement\_date=15,

            payment\_due\_date=10,

            annual\_interest\_rate=18.00,

            opening\_date=date.today() - timedelta(days=90)

        )

        self.test\_account.save()

        # Create the financial calculator

        self.calculator = FinancialCalculator()

    def tearDown(self):

        """Clean up after test"""

        db.close()

    def test\_minimum\_payment\_calculation(self):

        """Test minimum payment calculation"""

        # Test with various balances

        self.assertEqual(self.calculator.calculate\_minimum\_payment(100), 10)  # 10% of small balance

        self.assertEqual(self.calculator.calculate\_minimum\_payment(500), 50)  # 10% of balance

        self.assertEqual(self.calculator.calculate\_minimum\_payment(5000), 250)  # 5% for larger balance

        # Test minimum payment floor

        self.assertEqual(self.calculator.calculate\_minimum\_payment(50), 10)  # Minimum payment is 10

    def test\_interest\_calculation(self):

        """Test interest calculation"""

        # Create transactions for testing

        for i in range(1, 6):

            # Create purchase transactions across statement period

            days\_ago = 45 - (i \* 5)  # Spread transactions across statement period

            transaction\_date = date.today() - timedelta(days=days\_ago)

            transaction = Transaction(

                account\_id=self.test\_account.account\_id,

                transaction\_date=datetime.combine(transaction\_date, datetime.min.time()),

                posting\_date=datetime.combine(transaction\_date + timedelta(days=1), datetime.min.time()),

                merchant\_name=f"TEST MERCHANT {i}",

                amount=-500.00,  # 500 per transaction

                description=f"Test purchase {i}",

                transaction\_type="PURCHASE",

                currency="MYR"

            )

            transaction.save()

            transaction.update\_account\_balance()

        # Make a payment 20 days ago

        payment\_date = date.today() - timedelta(days=20)

        payment = Transaction(

            account\_id=self.test\_account.account\_id,

            transaction\_date=datetime.combine(payment\_date, datetime.min.time()),

            posting\_date=datetime.combine(payment\_date, datetime.min.time()),

            merchant\_name="CIMB PAYMENT",

            amount=1000.00,  # 1000 payment

            description="Test payment",

            transaction\_type="PAYMENT",

            currency="MYR"

        )

        payment.save()

        payment.update\_account\_balance()

        # Retrieve updated account

        self.test\_account = Account.get\_by\_id(self.test\_account.account\_id)

        # Generate a billing cycle

        today = date.today()

        prev\_month = today.replace(day=1) - timedelta(days=1)

        start\_date = prev\_month.replace(day=16)  # 16th of previous month

        end\_date = today.replace(day=15)  # 15th of current month

        cycle\_id = self.calculator.create\_billing\_cycle(

            self.test\_account.account\_id,

            start\_date,

            end\_date

        )

        # Calculate interest for the cycle

        interest = self.calculator.calculate\_interest(self.test\_account.account\_id, cycle\_id)

        # Expected balance: 5\*500 - 1000 = 1500

        # Expected interest at 18% APR for a month: ~1500 \* 0.18 / 12 = ~22.5

        # But actual calculation will vary based on daily balances

        self.assertGreater(interest, 0)

        # Test that interest is recorded in the database

        query = """

            SELECT \* FROM interest\_calculations

            WHERE account\_id = %s AND cycle\_id = %s

        """

        results = db.execute\_query(query, (self.test\_account.account\_id, cycle\_id), fetch=True)

        self.assertEqual(len(results), 1)

        self.assertEqual(float(results[0]['principal\_amount']), 1500.00)

    def test\_statement\_summary\_calculation(self):

        """Test statement summary calculations"""

        # Create initial balance

        self.test\_account.current\_balance = 1000.00

        self.test\_account.save()

        # Create some purchases and payments in the statement period

        today = date.today()

        statement\_start = today.replace(day=1)

        statement\_end = today.replace(day=15) if today.day > 15 else today

        # Create 3 purchases

        for i in range(1, 4):

            transaction\_date = statement\_start + timedelta(days=i \* 3)

            if transaction\_date <= statement\_end:

                transaction = Transaction(

                    account\_id=self.test\_account.account\_id,

                    transaction\_date=datetime.combine(transaction\_date, datetime.min.time()),

                    posting\_date=datetime.combine(transaction\_date + timedelta(days=1), datetime.min.time()),

                    merchant\_name=f"TEST MERCHANT {i}",

                    amount=-200.00,  # 200 per transaction

                    description=f"Test purchase {i}",

                    transaction\_type="PURCHASE",

                    currency="MYR"

                )

                transaction.save()

                transaction.update\_account\_balance()

        # Create 1 payment

        payment\_date = statement\_start + timedelta(days=10)

        if payment\_date <= statement\_end:

            payment = Transaction(

                account\_id=self.test\_account.account\_id,

                transaction\_date=datetime.combine(payment\_date, datetime.min.time()),

                posting\_date=datetime.combine(payment\_date, datetime.min.time()),

                merchant\_name="CIMB PAYMENT",

                amount=300.00,  # 300 payment

                description="Test payment",

                transaction\_type="PAYMENT",

                currency="MYR"

            )

            payment.save()

            payment.update\_account\_balance()

        # Create a billing cycle

        cycle\_id = self.calculator.create\_billing\_cycle(

            self.test\_account.account\_id,

            statement\_start,

            statement\_end

        )

        # Calculate statement summary

        summary = self.calculator.calculate\_statement\_summary(

            self.test\_account.account\_id,

            cycle\_id

        )

        # Check summary calculations

        self.assertEqual(summary['previous\_balance'], Decimal('1000.00'))

        # Check if number of purchases is correct

        # Note: This may vary based on today's date when running the test

        purchases\_in\_period = Transaction.get\_by\_date\_range(

            self.test\_account.account\_id,

            statement\_start,

            statement\_end,

            "PURCHASE"

        )

        self.assertEqual(summary['total\_purchases'], Decimal(abs(sum(float(t.amount) for t in purchases\_in\_period))))

        # Check if number of payments is correct

        payments\_in\_period = Transaction.get\_by\_date\_range(

            self.test\_account.account\_id,

            statement\_start,

            statement\_end,

            "PAYMENT"

        )

        self.assertEqual(summary['total\_payments'], Decimal(sum(float(t.amount) for t in payments\_in\_period)))

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

21.test\test\_database.py

# tests/test\_database.py

import unittest

import os

import sys

from datetime import datetime, date

# Add project root to path

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

from database.connection import db

from models.customer import Customer

from models.account import Account

from models.transaction import Transaction

class TestDatabaseConnection(unittest.TestCase):

    """Test the database connection and basic operations"""

    def setUp(self):

        """Set up test environment"""

        self.connection = db.connect()

        # Clear test data

        cursor = self.connection.cursor()

        cursor.execute("SET FOREIGN\_KEY\_CHECKS = 0")

        cursor.execute("TRUNCATE TABLE transactions")

        cursor.execute("TRUNCATE TABLE accounts")

        cursor.execute("TRUNCATE TABLE customers")

        cursor.execute("SET FOREIGN\_KEY\_CHECKS = 1")

        self.connection.commit()

        # Create test customer

        self.test\_customer = Customer(

            first\_name="Test",

            last\_name="User",

            email="test@example.com",

            phone="+60123456789",

            address\_line1="123 Test Street",

            city="Test City",

            state="Test State",

            postal\_code="12345",

            country="Malaysia",

            date\_of\_birth=date(1990, 1, 1)

        )

        self.test\_customer.save()

    def tearDown(self):

        """Clean up after test"""

        db.close()

    def test\_customer\_creation(self):

        """Test creating a customer"""

        # Retrieve the customer we created in setUp

        retrieved\_customer = Customer.get\_by\_email("test@example.com")

        self.assertIsNotNone(retrieved\_customer)

        self.assertEqual(retrieved\_customer.first\_name, "Test")

        self.assertEqual(retrieved\_customer.last\_name, "User")

    def test\_account\_creation(self):

        """Test creating an account"""

        # Create a test account

        test\_account = Account(

            customer\_id=self.test\_customer.customer\_id,

            account\_number="TEST0001234567",

            card\_number="5196061234567890",

            card\_type="MASTERCARD",

            credit\_limit=10000.00,

            current\_balance=0.00,

            statement\_date=15,

            payment\_due\_date=5,

            annual\_interest\_rate=17.90,

            opening\_date=date.today()

        )

        test\_account.save()

        # Retrieve the account

        retrieved\_account = Account.get\_by\_account\_number("TEST0001234567")

        self.assertIsNotNone(retrieved\_account)

        self.assertEqual(retrieved\_account.card\_type, "MASTERCARD")

        self.assertEqual(float(retrieved\_account.credit\_limit), 10000.00)

    def test\_transaction\_creation(self):

        """Test creating a transaction"""

        # Create a test account

        test\_account = Account(

            customer\_id=self.test\_customer.customer\_id,

            account\_number="TEST0001234567",

            card\_number="5196061234567890",

            card\_type="MASTERCARD",

            credit\_limit=10000.00,

            current\_balance=0.00,

            statement\_date=15,

            payment\_due\_date=5,

            annual\_interest\_rate=17.90,

            opening\_date=date.today()

        )

        test\_account.save()

        # Create a test transaction

        test\_transaction = Transaction(

            account\_id=test\_account.account\_id,

            transaction\_date=datetime.now(),

            posting\_date=datetime.now(),

            merchant\_name="TEST MERCHANT",

            amount=-150.00,

            description="Test purchase",

            transaction\_type="PURCHASE",

            currency="MYR"

        )

        test\_transaction.save()

        # Retrieve transactions for account

        transactions = Transaction.get\_by\_account\_id(test\_account.account\_id)

        self.assertEqual(len(transactions), 1)

        self.assertEqual(transactions[0].merchant\_name, "TEST MERCHANT")

        self.assertEqual(float(transactions[0].amount), -150.00)

    def test\_account\_balance\_update(self):

        """Test updating account balance with transactions"""

        # Create a test account

        test\_account = Account(

            customer\_id=self.test\_customer.customer\_id,

            account\_number="TEST0001234567",

            card\_number="5196061234567890",

            card\_type="MASTERCARD",

            credit\_limit=10000.00,

            current\_balance=0.00,

            statement\_date=15,

            payment\_due\_date=5,

            annual\_interest\_rate=17.90,

            opening\_date=date.today()

        )

        test\_account.save()

        # Create purchase transaction

        purchase\_transaction = Transaction(

            account\_id=test\_account.account\_id,

            transaction\_date=datetime.now(),

            posting\_date=datetime.now(),

            merchant\_name="TEST MERCHANT",

            amount=-500.00,

            description="Test purchase",

            transaction\_type="PURCHASE",

            currency="MYR"

        )

        purchase\_transaction.save()

        # Update account balance

        purchase\_transaction.update\_account\_balance()

        # Retrieve updated account

        updated\_account = Account.get\_by\_id(test\_account.account\_id)

        self.assertEqual(float(updated\_account.current\_balance), 500.00)

        # Create payment transaction

        payment\_transaction = Transaction(

            account\_id=test\_account.account\_id,

            transaction\_date=datetime.now(),

            posting\_date=datetime.now(),

            merchant\_name="CIMB PAYMENT",

            amount=200.00,

            description="Test payment",

            transaction\_type="PAYMENT",

            currency="MYR"

        )

        payment\_transaction.save()

        # Update account balance

        payment\_transaction.update\_account\_balance()

        # Retrieve updated account

        updated\_account = Account.get\_by\_id(test\_account.account\_id)

        self.assertEqual(float(updated\_account.current\_balance), 300.00)

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

22.test\test\_pdf\_generation

# tests/test\_pdf\_generation.py

import unittest

import os

import sys

from datetime import datetime, date, timedelta

import tempfile

# Add project root to path

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

from pdf\_generator.pdf\_builder import PDFBuilder

from pdf\_generator.template\_manager import TemplateManager

from models.customer import Customer

from models.account import Account

from models.transaction import Transaction

from processors.financial\_calculator import FinancialCalculator

from database.connection import db

class TestPdfGeneration(unittest.TestCase):

    """Test PDF generation for credit card statements"""

    def setUp(self):

        """Set up test environment"""

        self.connection = db.connect()

        # Clear test data

        cursor = self.connection.cursor()

        cursor.execute("SET FOREIGN\_KEY\_CHECKS = 0")

        cursor.execute("TRUNCATE TABLE transactions")

        cursor.execute("TRUNCATE TABLE accounts")

        cursor.execute("TRUNCATE TABLE customers")

        cursor.execute("TRUNCATE TABLE billing\_cycles")

        cursor.execute("TRUNCATE TABLE interest\_calculations")

        cursor.execute("SET FOREIGN\_KEY\_CHECKS = 1")

        self.connection.commit()

        # Create test customer and account

        self.test\_customer = Customer(

            first\_name="Test",

            last\_name="User",

            email="test@example.com",

            phone="+60123456789",

            address\_line1="123 Test Street",

            city="Test City",

            state="Test State",

            postal\_code="12345",

            country="Malaysia",

            date\_of\_birth=date(1990, 1, 1)

        )

        self.test\_customer.save()

        self.test\_account = Account(

            customer\_id=self.test\_customer.customer\_id,

            account\_number="TEST0001234567",

            card\_number="5196061234567890",

            card\_type="MASTERCARD",

            credit\_limit=10000.00,

            current\_balance=1000.00,

            statement\_date=15,

            payment\_due\_date=10,

            annual\_interest\_rate=18.00,

            opening\_date=date.today() - timedelta(days=90)

        )

        self.test\_account.save()

        # Create test transactions

        self.create\_test\_transactions()

        # Create a billing cycle for testing

        self.calculator = FinancialCalculator()

        today = date.today()

        statement\_month = today.replace(day=1) - timedelta(days=1) if today.day < 15 else today

        self.statement\_start = statement\_month.replace(day=16)  # 16th of previous month

        if today.day < 15:

            self.statement\_end = today.replace(day=15, month=today.month-1)  # 15th of previous month

        else:

            self.statement\_end = today.replace(day=15)  # 15th of current month

        self.cycle\_id = self.calculator.create\_billing\_cycle(

            self.test\_account.account\_id,

            self.statement\_start,

            self.statement\_end

        )

        # Calculate interest

        self.calculator.calculate\_interest(self.test\_account.account\_id, self.cycle\_id)

        # Update summary

        self.calculator.calculate\_statement\_summary(self.test\_account.account\_id, self.cycle\_id)

        # Create template manager and PDF builder

        self.template\_manager = TemplateManager()

        self.pdf\_builder = PDFBuilder(self.template\_manager)

    def tearDown(self):

        """Clean up after test"""

        db.close()

    def create\_test\_transactions(self):

        """Create test transactions for the statement period"""

        # Get dates for the statement period

        today = date.today()

        prev\_month = today.replace(day=1) - timedelta(days=1)

        start\_date = prev\_month.replace(day=16)  # 16th of previous month

        end\_date = today.replace(day=15)  # 15th of current month

        # Create some purchases

        for i in range(1, 6):

            transaction\_date = start\_date + timedelta(days=i \* 5)

            if transaction\_date <= end\_date:

                transaction = Transaction(

                    account\_id=self.test\_account.account\_id,

                    transaction\_date=datetime.combine(transaction\_date, datetime.min.time()),

                    posting\_date=datetime.combine(transaction\_date + timedelta(days=1), datetime.min.time()),

                    merchant\_name=f"TEST MERCHANT {i}",

                    merchant\_category="Shopping",

                    amount=-200.00,  # 200 per transaction

                    description=f"Test purchase {i}",

                    reference\_number=f"REF{i}",

                    transaction\_type="PURCHASE",

                    currency="MYR"

                )

                transaction.save()

                transaction.update\_account\_balance()

        # Create a payment

        payment\_date = start\_date + timedelta(days=20)

        if payment\_date <= end\_date:

            payment = Transaction(

                account\_id=self.test\_account.account\_id,

                transaction\_date=datetime.combine(payment\_date, datetime.min.time()),

                posting\_date=datetime.combine(payment\_date, datetime.min.time()),

                merchant\_name="CIMB PAYMENT",

                amount=500.00,  # 500 payment

                description="Test payment",

                reference\_number="PAYREF1",

                transaction\_type="PAYMENT",

                currency="MYR"

            )

            payment.save()

            payment.update\_account\_balance()

    def test\_template\_loading(self):

        """Test loading of the CIMB template"""

        template = self.template\_manager.get\_template("CIMB")

        self.assertIsNotNone(template)

        self.assertEqual(template.name, "CIMB")

    def test\_pdf\_generation(self):

        """Test generating a PDF statement"""

        # Create a temporary file for the PDF

        with tempfile.NamedTemporaryFile(suffix='.pdf', delete=False) as temp\_file:

            pdf\_path = temp\_file.name

        try:

            # Generate the PDF

            result = self.pdf\_builder.generate\_statement(

                self.test\_account.account\_id,

                self.cycle\_id,

                output\_path=pdf\_path

            )

            # Check if the PDF was generated

            self.assertTrue(result['success'])

            self.assertTrue(os.path.exists(pdf\_path))

            self.assertGreater(os.path.getsize(pdf\_path), 0)

            # Update the billing cycle with the PDF path

            query = """

                UPDATE billing\_cycles

                SET statement\_generated = 1, pdf\_file\_path = %s

                WHERE cycle\_id = %s

            """

            db.execute\_query(query, (pdf\_path, self.cycle\_id))

            # Verify the update

            query = "SELECT \* FROM billing\_cycles WHERE cycle\_id = %s"

            result = db.execute\_query(query, (self.cycle\_id,), fetch=True)

            self.assertEqual(result[0]['statement\_generated'], 1)

            self.assertEqual(result[0]['pdf\_file\_path'], pdf\_path)

        finally:

            # Clean up the temporary file

            if os.path.exists(pdf\_path):

                os.remove(pdf\_path)

    def test\_statement\_data\_preparation(self):

        """Test preparation of statement data"""

        # Get statement data

        statement\_data = self.pdf\_builder.prepare\_statement\_data(

            self.test\_account.account\_id,

            self.cycle\_id

        )

        # Check basic information

        self.assertEqual(statement\_data['account']['account\_number'], self.test\_account.account\_number)

        self.assertEqual(statement\_data['customer']['first\_name'], self.test\_customer.first\_name)

        self.assertEqual(statement\_data['customer']['last\_name'], self.test\_customer.last\_name)

        # Check billing cycle information

        self.assertEqual(statement\_data['billing\_cycle']['start\_date'], self.statement\_start)

        self.assertEqual(statement\_data['billing\_cycle']['end\_date'], self.statement\_end)

        # Check transactions

        self.assertGreater(len(statement\_data['transactions']), 0)

        # Check that transactions are within the statement period

        for transaction in statement\_data['transactions']:

            transaction\_date = transaction['transaction\_date'].date() if isinstance(transaction['transaction\_date'], datetime) else transaction['transaction\_date']

            self.assertTrue(self.statement\_start <= transaction\_date <= self.statement\_end)

if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

23.main.py

#!/usr/bin/env python3

"""

Credit Card Statement Generator

Main application entry point that coordinates the statement generation process.

"""

import os

import argparse

import logging

from datetime import datetime

from pathlib import Path

from utils.config import Config

from database.connection import DatabaseConnection

from models.customer import Customer

from models.account import Account

from models.transaction import Transaction

from processors.financial\_calculator import FinancialCalculator

from processors.data\_processor import DataProcessor

from pdf\_generator.pdf\_builder import PDFBuilder

from pdf\_generator.template\_manager import TemplateManager

#from utils.config import Config

#from utils.error\_handler import setup\_error\_handling

#from utils.language import get\_language\_text

config = Config('config.json')  # Pass the filename if needed

def setup\_logging():

    """Configure logging for the application."""

    log\_dir = Path("logs")

    log\_dir.mkdir(exist\_ok=True)

    log\_file = log\_dir / f"statement\_generator\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.log"

    logging.basicConfig(

        level=logging.INFO,

        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

        handlers=[

            logging.FileHandler(log\_file),

            logging.StreamHandler()

        ]

    )

    return logging.getLogger(\_\_name\_\_)

def parse\_arguments():

    """Parse command line arguments."""

    parser = argparse.ArgumentParser(description='Generate credit card statements')

    parser.add\_argument('--config', type=str, default='config.ini', help='Path to configuration file')

    parser.add\_argument('--customer-id', type=int, help='Generate statement for specific customer ID')

    parser.add\_argument('--account-id', type=int, help='Generate statement for specific account ID')

    parser.add\_argument('--statement-date', type=str, help='Statement date (YYYY-MM-DD)')

    parser.add\_argument('--output-dir', type=str, default='statements', help='Output directory for statements')

    parser.add\_argument('--bank-template', type=str, help='Specific bank template to use')

    parser.add\_argument('--language', type=str, default='en', help='Language for statements (e.g., en, es, fr)')

    parser.add\_argument('--test', action='store\_true', help='Run in test mode')

    return parser.parse\_args()

def generate\_statement(db\_conn, customer\_id, account\_id, statement\_date, output\_dir, bank\_template, language):

    """Generate a credit card statement for the specified account."""

    logger = logging.getLogger(\_\_name\_\_)

    # Get customer data

    customer = Customer.get\_by\_id(db\_conn, customer\_id)

    if not customer:

        logger.error(f"Customer {customer\_id} not found")

        return False

    # Get account data

    account = Account.get\_by\_id(db\_conn, account\_id)

    if not account:

        logger.error(f"Account {account\_id} not found")

        return False

    # Verify account belongs to customer

    if account.customer\_id != customer\_id:

        logger.error(f"Account {account\_id} does not belong to customer {customer\_id}")

        return False

    # Get transactions for the statement period

    transactions = Transaction.get\_for\_statement(db\_conn, account\_id, statement\_date)

    # Process financial data

    calculator = FinancialCalculator()

    processor = DataProcessor()

    statement\_data = processor.prepare\_statement\_data(

        customer,

        account,

        transactions,

        statement\_date

    )

    # Calculate financial summaries

    statement\_data = calculator.calculate\_statement\_totals(statement\_data)

    statement\_data = calculator.calculate\_interest(statement\_data)

    statement\_data = calculator.calculate\_minimum\_payment(statement\_data)

    # Apply language translations

    statement\_data['language\_text'] = get\_language\_text(language)

    # Generate PDF statement

    template\_manager = TemplateManager()

    template = template\_manager.get\_template(bank\_template or account.bank\_name)

    pdf\_builder = PDFBuilder(template)

    pdf\_path = pdf\_builder.generate\_pdf(statement\_data, output\_dir)

    logger.info(f"Generated statement for account {account\_id} at {pdf\_path}")

    return pdf\_path

def main():

    """Main entry point for the credit card statement generator."""

    # Setup

    logger = setup\_logging()

    args = parse\_arguments()

    # Load configuration

    config = Config(args.config)

    # Setup error handling

    setup\_error\_handling()

    # Create output directory if it doesn't exist

    output\_dir = Path(args.output\_dir)

    output\_dir.mkdir(exist\_ok=True, parents=True)

    try:

        # Initialize database connection

        db\_conn = DatabaseConnection(

            config.get('database', 'host'),

            config.get('database', 'port', type=int),

            config.get('database', 'database'),

            config.get('database', 'user'),

            config.get('database', 'password')

        )

        statement\_date = args.statement\_date or datetime.now().strftime("%Y-%m-%d")

        if args.test:

            logger.info("Running in test mode")

            # Run test generation with sample data

            from tests.test\_cases import get\_test\_cases

            test\_cases = get\_test\_cases()

            for test\_case in test\_cases:

                generate\_statement(

                    db\_conn,

                    test\_case['customer\_id'],

                    test\_case['account\_id'],

                    test\_case['statement\_date'],

                    args.output\_dir,

                    test\_case.get('bank\_template'),

                    args.language

                )

        elif args.customer\_id and args.account\_id:

            # Generate a specific statement

            generate\_statement(

                db\_conn,

                args.customer\_id,

                args.account\_id,

                statement\_date,

                args.output\_dir,

                args.bank\_template,

                args.language

            )

        else:

            # Generate statements for all accounts due for statements

            accounts\_due = Account.get\_accounts\_due\_for\_statements(db\_conn, statement\_date)

            logger.info(f"Generating statements for {len(accounts\_due)} accounts")

            for account in accounts\_due:

                generate\_statement(

                    db\_conn,

                    account.customer\_id,

                    account.id,

                    statement\_date,

                    args.output\_dir,

                    args.bank\_template or account.bank\_name,

                    args.language

                )

        logger.info("Statement generation completed successfully")

    except Exception as e:

        logger.exception(f"Error during statement generation: {e}")

        return 1

    finally:

        if 'db\_conn' in locals():

            db\_conn.close()

    return 0

import logging

from config import Config

def main():

    try:

        # Initialize Config object

        config = Config('config.json')  # Specify the config file if needed

        print(config.get\_config())  # Or do something with the config

    except ConfigurationError as e:

        logging.error(f"Failed to load configuration: {str(e)}")

        exit(1)

if \_\_name\_\_ == "\_\_main\_\_":

    exit\_code = main()

    exit(exit\_code)