Report: Project 5 Final

Comprehensive Report

Interest Calculations

1. Reward Calculation

- The calculation of accumulated rewards for 2025 is simply the sum of each of the respective categories multiplied by the percentage for that category.
- Groceries $7200 * .03 = $216, Gas $1200 * .02 = $24 , Other $3600 * .01 = $36
- The Carl's total rewards for 2025 are $276.

2. Simple Interest Calculation

- While not accurate, as interest is normally calculated with a daily average balance, you can make a rough estimate of the interest Carl pays for 2025. This is simply the annual interest rate divided by the number of months and then multiplied by the revolving balance. This amount is then compounded onto the previous balance for the next months calculation.
- 35% Annually is 2.92% monthly.
    - January interest and ending balance: $29.20 and $1029.20
    - February interest and ending balance: $30.05 and $1059.25
    - March interest and ending balance: $30.93 and $1090.18
    - April interest and ending balance: $31.83 and $1122.02
    - May interest and ending balance: $32.76 and $1154.78
    - June interest and ending balance: $33.72 and $1188.50
    - July interest and ending balance: $34.70 and $1223.20
    - August interest and ending balance: $35.72 and $1258.92
    - September interest and ending balance: $36.76 and $1295.68
    - October interest and ending balance: $37.83 and $1333.51
    - November interest and ending balance: $38.94 and $1372.45
    - December interest and ending balance: $40.08 and $1412.53
- The total accrued interest for 2025 is **$412.53**, this is considerably higher than the *$350 that someone could reasonably expect 35% interest to be*. This is due to the compounding interest.

3. Average Daily Balance Calculation

3.1. Daily Periodic Rate

- DPR = APR/360 (use of 360 as per project instructions)
    - DPR = .35/360 (use of 360 as per project instructions)
    - DPR = .000973

3.2. Average Daily Balance Compounding at the End of the Billing Cycle

Annual Interest Rate	35%
Daily Interest Rate	0.000972222
$1000 charged on the 15th of the month	
$1000 paid on the 20th of the month	
Interest compounded on the 30th of each month	

	1st to 15th	16th to 20th	20th to 30th	ADB For the month	Monthly Interest Payment
January					
Balance	$ 1,000.00	$ 2,000.00	$ 2,000.00		
Calculations	15000	10000	20000	$ 1,500.00	$ 43.75
February					
Balance	$ 2,043.75	$ 3,043.75	$ 2,043.75		
Calculations	30656.25	15218.75	20437.50	$ 2,210.42	$ 64.47
March					
Balance	$ 2,108.22	$ 3,108.22	$ 2,108.22		
Calculations	31623.31	15541.10	21082.20	$ 2,274.89	$ 66.35
April					
Balance	$ 2,174.57	$ 3,174.57	$ 2,174.57		
Calculations	32618.57	15872.86	21745.71	$ 2,341.24	$ 68.29
May					
Balance	$ 2,242.86	$ 3,242.86	$ 2,242.86		
Calculations	33642.86	16214.29	22428.57	$ 2,409.52	$ 70.28
June					
Balance	$ 2,313.14	$ 3,313.14	$ 2,313.14		
Calculations	34697.03	16565.68	23131.35	$ 2,479.80	$ 72.33
July					
Balance	$ 2,385.46	$ 3,385.46	$ 2,385.46		
Calculations	35781.94	16927.31	23854.63	$ 2,552.13	$ 74.44

August					
Balance	$ 2,459.90	$ 3,459.90	$ 2,459.90		
Calculations	36898.50	17299.50	24599.00	$ 2,626.57	$ 76.61

September					
Balance	$ 2,536.51	$ 3,536.51	$ 2,536.51		
Calculations	38047.62	17682.54	25365.08	$ 2,703.17	$ 78.84

October					
Balance	$ 2,615.35	$ 3,615.35	$ 2,615.35		
Calculations	39230.26	18076.75	26153.51	$ 2,782.02	$ 81.14

November					
Balance	$ 2,696.49	$ 3,696.49	$ 2,696.49		
Calculations	40447.39	18482.46	26964.93	$ 2,863.16	$ 83.51

December					
Balance	$ 2,780.00	$ 3,780.00	$ 2,780.00		
Calculations	41700.03	18900.01	27800.02	$ 2,946.67	$ 85.94

Total Interest for 2025	*$*	*865.95*
Ending balance for 2025	*$*	*2,780.00*

4. Synchrony Bank Balance Calculation

4.1 Calculations

Annual Interest Rate	35%
Daily Interest Rate	0.000972222
$1000 charged on the 15th of the month	
$1000 paid on the 20th of the month	
Interest compounded daily	

January		
Balance	$	2,044.26
Interest	$	44.26
February		
Balance	$	2,109.66
Interest	$	65.39
March		
Balance	$	2,176.98
Interest	$	67.33
April		
Balance	$	2,246.30
Interest	$	69.32
May		
Balance	$	2,317.66
Interest	$	71.37
June		
Balance	$	2,391.14
Interest	$	73.48
July		
Balance	$	2,466.79
Interest	$	75.65

Day	Dec	Balance
		$2,792.45
1	$ 2.71	$2,795.17
2	$ 2.72	$2,797.89
3	$ 2.72	$2,800.61
4	$ 2.72	$2,803.33
5	$ 2.73	$2,806.05
6	$ 2.73	$2,808.78
7	$ 2.73	$2,811.51
8	$ 2.73	$2,814.25
9	$ 2.74	$2,816.98
10	$ 2.74	$2,819.72
11	$ 2.74	$2,822.46
12	$ 2.74	$2,825.21
13	$ 2.75	$2,827.95
14	$ 2.75	$2,830.70
15	$ 2.75	$3,833.45
16	$ 3.73	$3,837.18
17	$ 3.73	$3,840.91
18	$ 3.73	$3,844.65
19	$ 3.74	$3,848.38
20	$ 3.74	$2,852.13
21	$ 2.77	$2,854.90
22	$ 2.78	$2,857.67
23	$ 2.78	$2,860.45
24	$ 2.78	$2,863.23
25	$ 2.78	$2,866.02
26	$ 2.79	$2,868.80
27	$ 2.79	$2,871.59
28	$ 2.79	$2,874.38
29	$ 2.79	$2,877.18
30	$ 2.80	$2,879.98
Interest Sum:	$ 87.52	

August		
Balance	$	2,544.68
Interest	$	77.89

September		
Balance	$	2,624.88
Interest	$	80.19

October		
Balance	$	2,707.44
Interest	$	82.57

November		
Balance	$	2,792.45
Interest	$	85.01

December		
Balance	$	2,879.98
Interest	$	87.52

Total Interest for 2025	*$*	*879.97*
Ending balance for 2025	*$*	*2,879.98*

4.2 Fundamental Differences

- Comparing the first three months of the Synchrony Bank calculations versus the Average Daily Balance method shown previously, we can see that the interest accrues much quicker when compounded daily, versus at the end of the billing cycle. Using the Average Daily Balance method, Carl would have paid $174.57 in interest over the first quarter, while having paid $176.98 in intertest as calculated by Synchrony Bank. This may not seem like a lot, at face value; however, over the course of one year, you would pay approximately 2% more interest.

5. Test Cases

- If implemented correctly, the only difference in result of a test case would be based on when the interest is compounded. For the Average Daily Balance method, I would still calculate the interest each day, but I would sum it up and compound it at the end of the month, as opposed to daily for the Synchrony Bank method. This will make implementation much easier, as well as testing.
- Regarding the specifics of the test, the parameters needed would be the length of the period (I assume it will be one month), previous balance, the transactions during the period and which method for interest calculation is being used. The interest rate would be fixed in the program or possibly set via the graphical user interface.

- For a test of 1 month, with a previous balance of $2000, charges of $1000 on the 15th, payment of $1000 on the 20th, the result should be $34.03 for the Average Daily Balance method and $34.50 for the Synchrony Bank method.
- Due to the usage of floating-point numbers, it is advisable to avoid relying on exact values for test results and instead establish appropriate upper and lower bounds.

Software Design

My Prompt

create a prompt that a software architect would use to create a model-view-controller program with the following: Stack: Java using the Swing/AWT graphical frontend, standalone on a MacOS, Microsoft Windows PC or Laptop, use JUnit 5 for unit tests, any libraries freely available, well documented with Javadocs and safe to use. the Use Case will be the attached SynchronyBankAccountingAgreement.pdf, UseCaseDescription.pdf and the following: Let's work with an example: Carl is our example credit card customer who has a cash back credit card that he uses for everyday purchases. The card incentivizes certain purchases with cash back such as 3% for groceries at supermarkets (Groceries), 2% at gas stations / ev charging stations (Gas), and 1% for all other merchant categories (Other). Cash back can be redeemed in various ways and Carl chooses to use it for gift cards. Carl loves this and uses the card regularly, but does this economically work well for him? Can you figure how much Carl is earning in rewards, paying for interest and fees over a year that goes like this:- He starts the year of 2025 with a leftover balance of $1000 on Jan 1 after he paid everything else on New Year's Eve and the account is in good standing. It's just that he has a revolving balance of $1000 from 2024 which implies interest charges.- He purchases groceries for $600, fills his car for $100, and pays for various other expenses $300 with this card each month. Total expenses per month: $1000.- He pays each month $1000 on the due date, which is the 20th of the month on time such that he does not incur any late fees and his account is in good standing.- His first payment for 2025 is on February 20 as he paid his Dec 31, 2024 closing statement early on the last day of 2024.- The APR is 35%.- The billing cycle starts on the 1st of the month and ends on the last day of the month.- The due date is the 20th of the month. - The account has a grace period if a customer pays the balance in full. Carl does not do this, so this unfortunately does not apply to Carl. He is charged interest on a regular basis. Simplifying assumptions to make the example calculations easier:- Assume each month has 30 days, the year has 360 days.- Assume all purchases take place on the 15th of the month, which is half way through the billing cycle. The purchase data will be in the attached Carl2025.csv file. The Model in the Model View Controller pattern should include a Transaction class to represent individual purchases or payments and an Account class to have the transaction data that is loaded from a file. Use a Strategy pattern to allow for different payment strategies. Use a Strategy pattern to allow for different variants of interest calculation algorithms. Use an Iterator pattern to go through transactions. Have the following four payment strategies: - Early Transactor: always pay the full statement balance a day after the cycle ends avoiding fees and interest. The starting balance is always $0. Occasionally uses rewards for gift cards, never for statement credit.- Wall Street Transactor: always pay the full statement balance at the due date, the last possible day avoiding fees and interest. Uses rewards each month for statement credit (i.e., as a partial payment that reduces the statement balance). The starting balance is always $0.- Light Revolver: pays five months the minimum payment each month and then pays after the 6th month the full statement

balance. After that it repeats. All payments are on the due date. In this way, the account is always in good standing, there are no late fees but interest accumulates till the account is paid in full. The starting balance can be selected and is usually larger than zero. The customer uses the rewards for gift cards, never for statement credit.- Heavy Revolver: only pays the minimum payment each month. Pays 5 months on time on the due date, then pays 1 month 5 days late incurring late fees and an increased penalty APR. After that it repeats. The starting balance can be selected and is usually in the thousands of dollars. The customer uses the rewards for gift cards, never for statement credit. Note that rewards are contingent on the account being in good standing.

Generated Prompt

Architect-Level Prompt for Generating an MVC Java Application

ROLE: You are a senior software architect designing a complete, production-quality Model-View-Controller (MVC) Java application with Swing/AWT UI.
GOAL: Generate the full codebase, build scripts, documentation, UML diagrams, and unit tests that implement the following system.

---

1. TECHNOLOGY STACK & QUALITY REQUIREMENTS

Language & Runtime

- Java 17 or later
- Stand-alone desktop application
- Must run on **macOS and Windows** (no OS-specific dependencies)

UI

- **Swing/AWT** graphics toolkit
- Classic MVC pattern (Views must not contain business logic)

Testing

- **JUnit 5**
- Mocking library allowed (e.g., Mockito)

Documentation

- Full **Javadoc** comments for all public classes and methods
- UML diagrams:
    - Class diagram
    - Sequence diagram for interest calculation
    - Package structure diagram

Build & Packaging

- Maven or Gradle (architect may choose)

- Runnable JAR artifact

Security & Safety

- No unsafe reflection
- No deprecated APIs
- Robust exception handling
- Safe parsing of user-provided CSV data

---

2. DOMAIN INPUT FILES

The system must process three external files provided by the user:

1. **SynchronyBankAccountAgreement.pdf**
   – Governs APR, penalty APR, minimum payment formula, fee rules, grace period logic, and interest calculation.
2. **UseCaseDescription.pdf**
   – Defines UI workflow, error handling, variation scenarios, visualization requirements.
3. **Carl2025.csv**
   – Contains 2025 transaction data for "Carl," formatted as:
4. yyyy-mm-dd, category, amount

Categories: *Groceries*, *Gas*, *Other*.

---

3. MAIN USE CASE & DOMAIN STORY

Implement the complete scenario:

Carl has a credit card with:

- **Rewards:**
  - 3% Groceries
  - 2% Gas/EV charging
  - 1% Other
- **Starting balance:** $1000 on Jan 1, 2025 (in good standing)
- **APR:** 35% (daily rate = APR / 365)

- **Penalty APR:** 39.99%
- **Billing cycle:** 1st → 30th of each month (assume 30-day months)
- **Due date:** 20th of the following month
- **Purchases:** All made on the 15th of each month
- **Monthly purchases:**
    - $600 groceries
    - $100 gas
    - $300 other
- **Payments:**
    - Pays $1000 monthly on Time (Feb 20 first payment)
    - Always in good standing
- **Rewards redemption:** Gift cards (no statement credit)
- **CSV file includes actual transaction rows; program must load these.**

The system must compute:

- Total rewards earned
- Total fees
- Total interest
- Ending balance
- Balance/time plots for visualization

---

4. SOFTWARE DESIGN REQUIREMENTS

4.1 Model Layer (Strict Business Logic)

Core Classes

Transaction

- Fields:
    - LocalDate date
    - enum Category { GROCERIES, GAS, OTHER }

    - BigDecimal amount
    - boolean isPayment
- Immutable
- Must include validation

Account

- Holds:
    - List<Transaction> (maintained in date order)
    - Starting balance
    - APR, penalty APR
    - Minimum payment formula (from agreement)
    - Calculated fields: interest, fees, rewards
- Methods:
    - loadFromCsv(File)
    - calculateMonthlyStatement()
    - applyPaymentStrategy(PaymentStrategy)
    - computeRewards()
    - computeInterest(InterestCalculationStrategy)
    - iterator() → Iterator<Transaction>

Design Patterns

- **Strategy Pattern** for:
    1. **PaymentStrategy**
        - EarlyTransactorStrategy
        - WallStreetTransactorStrategy
        - LightRevolverStrategy
        - HeavyRevolverStrategy
    2. **InterestCalculationStrategy**
        - AverageDailyBalanceStrategy

        - SynchronyDailyBalanceStrategy (as defined in PDF)
- **Iterator Pattern**
    - Account implements Iterable<Transaction>
    - Custom TransactionIterator for date-sorted traversal

---

5. CONTROLLER LAYER

Responsibilities

- Coordinate file loading
- Validate CSV
- Manage selected strategy objects
- Run calculations
- Populate view-model objects for presentation
- Handle all errors & user messages
- Invoke charts/visualizations (Swing Canvas, JFreeChart allowed)

Specific Behaviors

- For errors: file not found, CSV format error → show dialog & return user to file selection
- For parameter customization (variation #2)
- For strategy selection (variation #3)
- For detailed visualization (variation #4)

---

6. VIEW LAYER (Swing)

Screens

1. **Main Window**
    - Instructions
    - File selection button
    - Strategy selection dropdown

- Settings panel (APR, Starting Balance, Dates, etc.)

2. **Summary Panel**
    - Beginning balance
    - Ending balance
    - Total interest
    - Total fees
    - Total payments
    - Total rewards
    - Export to CSV option
3. **Visualization Window**
    - Daily/weekly/monthly/yearly graphs
    - Balance vs. time
    - Interest accrual over time
    - Fees timeline

Constraints

- Views must not perform domain calculations
- All UI actions must delegate to controllers

---

7. TESTING REQUIREMENTS (JUnit 5)

Create thorough tests for:

Transaction Tests

- Valid category
- Correct amount scaling
- Payment vs purchase logic

Account Tests

- Loading valid/invalid CSV
- Correct reward computation per category

- Correct interest computation under:
    - No grace period
    - Revolving balance
    - Penalty APR
- Correct fee assessment

Strategy Tests

- Minimum payment logic
- Full-balance payment
- 5-month revolve cycle
- Heavy revolver late fee & penalty APR trigger
- Early vs Wall Street Transactor differences
- Statement credit logic

Mocking

- Mock InterestCalculationStrategy and PaymentStrategy to verify controller behavior

---

8. OUTPUT REQUIREMENTS

The generated codebase must include:

Source Code

- /src/main/java
- /src/test/java

Resources

- Sample CSV
- Default configuration file (JSON or properties)

Documentation

- Javadocs (full)
- README with build/run instructions
- UML diagrams

- Architectural overview document explaining design patterns
- State machine for Heavy Revolver's penalty APR cycle

Deliverables that must be automatically generated

- All classes
- All interfaces
- All tested behaviors
- All diagrams
- Fully runnable Swing application

---

9. SUMMARY OF WHAT YOU MUST PRODUCE

Generate the **complete MVC Swing Java application**, including:

- Domain model
- Strategy pattern implementations
- Iterator pattern
- Controller logic
- Swing views
- Visualization component
- CSV loader
- Synchrony-compliant interest algorithm
- Error handling
- Four payment strategies
- All tests
- Javadocs
- UML diagrams
- Build scripts
- Runnable JAR configuration

Project: CreditCardSimulator Design 1.0

---

1 — Executive summary

A desktop Java 17+ Swing/AWT MVC application that loads a transaction CSV (Carl2025.csv), models an Account and Transactions, and simulates interest, fees, rewards and payment behavior across four payment strategies. Interest is calculated using a daily-balance algorithm following the Synchrony agreement; the UI offers file selection, strategy/settings, summary and visualizations. The app is test-covered with JUnit 5. Key business rules and numeric parameters are taken from the uploaded Synchrony agreement and Use Case document.

---

2 — High-level architecture & packages

com.creditcardsim

```
├── app              // main entry + bootstrap + DI (simple factory)
├── model            // domain objects & business logic
│   ├── account
│   ├── transaction
│   ├── strategy     // PaymentStrategy, InterestCalculationStrategy
│   └── iterator
├── controller       // controllers that mediate between Model & View
├── view             // Swing views, dialogs, charts
├── util             // CSV parsing, date helpers, money utils
├── config           // default settings, constants (json/properties)
└── test             // JUnit 5 tests mirroring packages above
```

---

3 — Domain model (classes, key fields & responsibilities)

Note: BigDecimal for currency, LocalDate for dates. All monetary operations use MathContext with scale=2 and rounding HALF_UP.

3.1 Transaction (immutable)

```
package com.creditcardsim.model.transaction;


public final class Transaction {
    public enum Category { GROCERIES, GAS, OTHER }


    private final LocalDate date;
    private final Category category;
    private final BigDecimal amount; // positive for purchases, negative for payments
    private final boolean isPayment; // derived: amount.signum() < 0


    // getters, validation in constructor, toString, equals/hashCode
}
```

3.2 Account

```
package com.creditcardsim.model.account;


public class Account implements Iterable<Transaction> {
    private final List<Transaction> transactions; // date-ordered
    private BigDecimal startingBalance;
    private BigDecimal apr;         // annual percentage (e.g., 0.35)
    private BigDecimal penaltyApr;  // (e.g., 0.3999)
    private int cycleStartDay;      // typically 1
    private int cycleLengthDays;    // 30 (per simplification)
    private int dueDayOfMonth;      // 20
    private Duration gracePeriod;   // derived (e.g., close->due)
    private boolean paperStatement; // for paper statement fee logic
    private BigDecimal paperStatementFee; // $1.99
```

```
    private BigDecimal minInterestCharge; // $2.00 minimum when interest > 0


    // Holds computed results (per-cycle)
    private List<Statement> statements;


    // public API:
    static Account loadFromCsv(Path csvFile, AccountConfig config) throws ParseException;
    void applyPaymentStrategy(PaymentStrategy strategy, InterestCalculationStrategy
interestCalc);
    Statement calculateStatement(YearMonth ym, InterestCalculationStrategy strategy);
    BigDecimal computeRewards(YearMonth ym);
    Iterator<Transaction> iterator();
    // getters and helper accessors
}
```

3.3 Statement (value object)

- Fields: YearMonth period, openingBalance, newCharges, payments, interestCharged, feesCharged, rewardsEarned, closingBalance, minimumPayment.
- Used for UI and export.

---

4 — Key interfaces & strategy patterns

4.1 PaymentStrategy

```
package com.creditcardsim.model.strategy;


public interface PaymentStrategy {
    /**
     * Returns list of Payment transactions to be scheduled for the given billing cycle statement.
     * Accepts Account snapshot + Statement and returns 0..n payments with dates & amounts.
```

```
    */
    List<Transaction> generatePayments(AccountSnapshot accountSnapshot, Statement
statement);
}
```

Implementations:

- EarlyTransactorStrategy — pays full statement balance 1 day after cycle ends. Starting balance 0 by default, rewards -> gift cards (no statement credit).
- WallStreetTransactorStrategy — pays full statement on due date (last day of grace period). Uses rewards as statement credit (partial payment).
- LightRevolverStrategy — pay minimum for 5 cycles, full balance on 6th, repeats. Payments on due date. Uses rewards for gift cards.
- HeavyRevolverStrategy — pay minimum each month; every 6th month pay 5 days late (late fee + triggers penalty APR after 2 occurrences within 12 cycles logic). Uses rewards for gift cards. Needs to model "late count" and penalty APR lifecycle.

4.2 InterestCalculationStrategy

```
package com.creditcardsim.model.strategy;


public interface InterestCalculationStrategy {
    /**
     * Given daily balances and applicable rates, compute interest for the billing cycle
     * and also return daily interest accrual details for visualization.
     */
    InterestResult computeInterest( AccountSnapshot snapshot, LocalDate cycleStart, LocalDate
cycleEnd );
}
```

Implementations:

- AverageDailyBalanceStrategy — classic avg. daily balance (optional)
- SynchronyDailyBalanceStrategy — implements "daily balance" exactly as laid out in Synchrony doc:

- daily balance = previous day starting_balance + charges - payments +/- fees
- dailyInterest = dailyBalance * (APR / 365)
- interest for cycle = sum(dailyInterest) but minimum $2 if any interest owed
- interest is added to balance each day for subsequent days (compounding as per doc). See algorithm section below.

---

5 — Iterator pattern

Account implements Iterable<Transaction> and returns TransactionIterator that yields transactions in chronological order. This iterator will be used by controllers and tests to iterate day-by-day or by grouping for monthly summaries.

---

6 — Important business rules (extracted from Synchrony agreement)

- **Daily rate:** APR / 365. Synchrony statement shows daily rates for APR 34.99% -> 0.0009587 (approx). Use exact APR parameter from config; for policy references see Synchrony PDF.
- **Interest accrual:** Interest accrues from purchase date until purchase is paid in full — daily-balance algorithm applies; interest is computed separately per balance type and added. Minimum interest charge = $2 in any cycle where interest > 0.
- **Minimum payment formula:** greater of ($30, 3.5% of new balance, or 1% plus interest & late fees) plus any past due amounts; round up to whole dollar; never more than new balance. Implement exactly as in the agreement.
- **Late fee:** $30 or $41 depending on prior 6-cycle history (for simplicity defaults can be $41 if any prior late). We will implement the two-tier behavior.
- **Paper statement fee:** $1.99 per month when balance > $2.50 and paper is selected.
- **Penalty APR trigger:** penalty APR applies if minimum payment missed two or more times during any 12 consecutive cycles; when in effect it may apply indefinitely. Implementation must track history and apply penalty APR to subsequent daily interest calculation.

(Design note: we will expose these parameters in config so devs/testers can tweak.)

---

7 — Daily interest algorithm (SynchronyDailyBalanceStrategy) — pseudo-code / deterministic steps

Inputs: cycleStartDate (inclusive), cycleEndDate (inclusive), startingBalance (balance at cycle start), list of transactions with dates within the cycle and possibly payments/credits, applicable APR for each balance type, dailyPenaltyRate if penalty applies.

Algorithm (day-by-day):

1. dailyRate = apr.divide(BigDecimal.valueOf(365), MC)
2. balanceForDay = balanceStartOfDay
   For day = cycleStartDate .. cycleEndDate:
   - apply all transactions that occur **on** this day (charges add, payments subtract), also apply fees (paper, late etc.) as transactions (treat fees as new purchases per agreement). (Important: sorting of postings if both payment and charge same day — treat charges posted before payments? Synchrony says starting balance + new charges - payments -> we'll process charges then payments.)
   - compute dailyInterest = balanceForDay.multiply(dailyRate) (if balanceForDay > 0). Use scale=10 for intermediate, sum to monthlyInterest.
   - set balanceForNextDay = balanceForDay.add(dailyInterest) (interest added to balance for next day compounding).
3. After loop, interestForCycle = sum(dailyInterests). If interestForCycle > 0 and interestForCycle < minInterestCharge then set interestForCycle = minInterestCharge (and treat the difference as a new purchaselike fee added to balance).
4. Return InterestResult with dailyInterest series and cycle total. Apply interest to account closing balance.

Edge cases & clarifications implemented:

- If account had balance at cycle start and was not paid in prior cycle in full, interest is charged from day 1 on purchases as described in the PDF.
- Promotional balances are out of scope for now but code path supports multiple balance types.

---

8 — Payment scheduling semantics (controller responsibilities)

- Controller orchestrates applying payment strategy for each statement.
- For each statement period:

1. compute newCharges (based on transactions)
2. compute minimumPayment via agreement formula
3. ask strategy for payments to schedule; strategy returns Transaction(s) dated appropriately
4. apply payments to account (the next cycle's daily balances will account for payments)
5. compute interest via interest strategy
6. compute fees (late, paper) as needed and append to transaction list for that cycle
7. mark if penalty APR must be set based on missed payments
8. prepare Statement object

- Controller persists Statements to a ResultModel to be displayed in View.

---

9 — UI (views) — screens / interactions

Main window (single frame)

- Top: Menu (File -> Open CSV, Settings, Export)
- Left panel: File loader + file metadata (#transactions, first/last date, total purchases)
- Center: Strategy selector (dropdown), parameter overrides (APR, starting balance, due day, paper statement flag)
- Right: Quick summary tiles (Opening balance, Ending balance, Total interest, Total fees, Total rewards)
- Bottom: Action buttons [Simulate] [Export Results] [Visualize]

Visualization window

- Tabs: Daily / Monthly / Yearly
- Charts:
  - Balance vs Time
  - Cumulative interest vs time
  - Fees vs time

- Table view for Statement list (month, opening, closing, interest, fees, payments, rewards)
- Use JFreeChart allowed (freely available) or custom Swing painting.

UI constraints:

- All heavy calc performed in background worker thread (SwingWorker) with progress bar (controller manages). (Note: you asked me earlier not to ask for waiting design — this is just design; actual implementation uses SwingWorker.)

---

10 — CSV format & parser behavior

CSV expected: yyyy-MM-dd,Category,Amount where Category ∈ {Groceries,Gas,Other} and Amount is positive for purchases (payments appear as negative amounts or flagged with category PAYMENT optional). The UseCase doc specifies this format. The loader will:

- validate date format ISO_LOCAL_DATE
- validate category list
- reject lines with invalid numbers or dates and report count of bad lines with sample rows
- create Transaction objects and sort them by date

(Developer note: the uploaded Carl2025.csv will be used for example runs and tests.)

---

11 — Testing plan (JUnit 5)

Unit test suites

- TransactionTest — immutability, parsing, validation, negative/positive amounts
- CsvLoaderTest — valid CSV, invalid CSV, missing file
- InterestCalculationStrategyTest:
    - Synchrony algorithm: known small scenario with deterministic daily balances — assert interest sum and min-interest behavior ($2 minimum). Use a tiny synthetic cycle.
- PaymentStrategyTest:
    - EarlyTransactor: given statement, ensure payment scheduled 1 day after cycle end and payment amount equals statement balance

  - WallStreetTransactor: payment scheduled on due date
  - LightRevolver: verify 5 minimums then full payment
  - HeavyRevolver: ensure late fee and penalty APR triggers after configured missed payment thresholds

- AccountIntegrationTest — run full-year simulation using Carl2025 sample and assert totals (rewards, interest, fees) are within expected tolerance. (We will include an "oracle" CSV of expected outcomes for regression.)
- ControllerTest — with mocked strategies/interest calculator (Mockito), verify controller behavior: schedules payments and populates Statement list, triggers penalty flag when required.

Test data

- small synthetic CSVs for unit tests
- Carl2025.csv used for integration tests

---

12 — Example class skeletons (selected)

InterestResult (value object)

```
public final class InterestResult {
    private final BigDecimal totalInterest;
    private final List<BigDecimal> dailyInterests; // aligned to cycle days
    // getters, builder
}
```

AccountSnapshot

Lightweight snapshot used by strategies to compute payments without mutating account state.

---

13 — Sequence diagram (textual) — Simulate one billing cycle

1. User clicks [Simulate] -> Controller
2. Controller -> CsvLoader: load transactions
3. Controller -> Account: loadFromCsv + set starting balance

4. Controller -> InterestCalculationStrategy: compute interest for cycleStart..cycleEnd
5. Controller -> PaymentStrategy: generatePayments(snapshot, statement)
6. Controller -> Account: apply payments (append transactions)
7. Controller -> Account: compute statement (interest + fees)
8. Controller -> View: publish Statement & update charts

---

14 — Persistence & export

- Results export: CSV summarizing statements and per-day series (balance, interest).
- Save/Load configuration: JSON config in user home (starting balance, APR override, rewards percents).
- No DB required.

---

15 — Logging & error handling

- Use SLF4J + Logback (or Java logging) with levels: INFO, WARN, ERROR.
- All user-facing errors surface via modal dialogs with short message + details log for developers.

---

16 — Documentation deliverables (initial)

- README.md with build/run instructions (Maven/Gradle), how to run runnable jar on macOS/Windows.
- Javadoc for all public classes (gradle/maven javadoc task)
- Architecture.md describing packages, patterns used
- Sequence and UML class diagrams (drawn with PlantUML); include PlantUML .puml files in /docs/diagrams
- Test report instructions

---

17 — Build & dependencies

- Build tool: **Maven** (recommended) or Gradle.

- Dependencies (all Maven central, freely available):
    - JUnit 5 (test)
    - Mockito (test)
    - JFreeChart (optional, visualization)
    - commons-csv or OpenCSV (CSV parsing)
    - SLF4J + Logback
- Produce fat runnable JAR (maven-shade).

---

18 — Milestones / next steps (developer-oriented)

1. Create project skeleton (maven) and packages.
2. Implement Transaction, CsvLoader, Account data structures, and unit tests for parsing.
3. Implement SynchronyDailyBalanceStrategy and unit tests with deterministic scenarios (assert interest/ min interest).
4. Implement PaymentStrategy interfaces + Early and WallStreet strategies first.
5. Implement simple Swing UI that loads CSV and displays summary tiles (no charts yet).
6. Add Light/Heavy Revolver strategies and penalty logic; add tests for penalty APR triggers.
7. Add charts and visualization; polish UI.
8. Document project and generate javadocs.

---

19 — Open design decisions & assumptions (I made these for the initial design)

- Month length: use 30-day month and 360-day year only for the simplified example (but engine supports real calendar days if config toggled). This matches your simplifications and the Use Case.
- Transactions that happen on the same day: apply charges first, then payments, then compute interest for that day (this decision matches the "starting balance + new charges - payments" wording). If you prefer payments first, we can flip the order.

- Rewards redemption: modeled as either "gift-card" (no payment effect) or "statement credit" (reduces statement balance). Wall Street uses statement credit; others use gift cards per your spec.
- Penalty APR: implemented per agreement (trigger: missed minimum >= 2 in rolling 12-month window). When applied, use penalty APR for subsequent interest until conditions to lift it are met — we add a method to attempt re-evaluation periodically.

---

20 — Example unit test scenario (pseudo JUnit)

```
@Test
void synchronyDailyBalance_minimumInterestApplied() {
   Account account = new Account(... apr = BigDecimal.valueOf(0.35), minInterestCharge = BigDecimal.valueOf(2));
   // starting balance $10, single day cycle with $0 payments
   InterestResult r = new SynchronyDailyBalanceStrategy().computeInterest(snapshot, LocalDate.of(2025,1,1), LocalDate.of(2025,1,30));
   assertTrue(r.getTotalInterest().compareTo(BigDecimal.ZERO) > 0);
   assertEquals(BigDecimal.valueOf(2.00).setScale(2), r.getTotalInterest()); // minimum interest override
}
```

---

21 — Files used for rules and UI guidance

- Synchrony credit card account agreement — interest formulas, min interest, minimum payment calculation, fees and penalty APR are used exactly as the authoritative rules.
- UseCaseDescription — UI flow, CSV format, allowed categories, and strategy definitions used to shape the controllers and views.

(Developer note: the Carl2025.csv you attached will be used as input when implementing and running integration tests.)

---

22 — Deliverables I can produce next (choose one)

- Full skeleton code (Maven project + package structure + base classes + tests stubs)

- Implementation of SynchronyDailyBalanceStrategy + unit tests
- Swing front-end starter (file loader + summary panel)
- PlantUML class + sequence diagrams exported as text files

Tell me which of the above you want implemented first and I will produce the code and tests for that piece right away.

Manual Changes

- Removed the GRADLE/MAVEN build systems
- Changed the project name and associated package names

Project: CreditCardSimulator Design 1.1 — Eclipse IDE Build System

This update modifies:

- Build process
- Directory layout
- Dependency management
- JUnit configuration
- External library handling
- Javadoc generation
- Packaging into an executable JAR

Everything else (architecture, MVC layers, strategies, algorithms, UI, controller behavior) remains exactly the same.

---

1 — Eclipse Build System Requirements

The project is now a standard **Eclipse Java Project**, *not* a Maven/Gradle project.

Eclipse Features Used

- Eclipse Java Build Path
- User Libraries for 3rd-party JAR management
- bin/ (or build/) as Eclipse-generated class output
- Export → Runnable JAR
- Eclipse JUnit 5 integration
- Eclipse Javadoc export

No pom.xml, no build.gradle.

All .classpath and .project files will be Eclipse-managed.

---

2 — Project Structure (Eclipse Format)

CreditCardSimulator/

```
├── src/                    // Java source code (default Eclipse "Source Folder")
│   ├── com.creditcardsim.app
│   ├── com.creditcardsim.model.account
│   ├── com.creditcardsim.model.transaction
│   ├── com.creditcardsim.model.strategy
│   ├── com.creditcardsim.model.iterator
│   ├── com.creditcardsim.controller
│   ├── com.creditcardsim.view
│   ├── com.creditcardsim.util
│   └── com.creditcardsim.config
├── test/                   // JUnit 5 Test Source Folder (must add manually)
│   ├── com.creditcardsim.*
├── lib/                    // EXTERNAL JARs manually added here
│   ├── junit-platform-console-standalone-x.y.z.jar
│   ├── mockito-core-x.y.z.jar
│   ├── jfreechart-x.y.z.jar (optional)
│   ├── opencsv-x.y.z.jar or commons-csv-x.y.z.jar
│   ├── slf4j-api.jar
│   ├── slf4j-simple.jar (optional)
│   └── any other required JARs
├── resources/              // non-code resources (icon, sample CSVs)
├── docs/                   // UML diagrams, documentation
└── CreditCardSimulator.launch // optional Eclipse run config
```

---

3 — Eclipse Project Configuration Steps

3.1 Create the Project

1. File → New → Java Project
2. Project name: **CreditCardSimulator**
3. Execution environment: **JavaSE-17**
4. Create module-info.java? → **Do NOT create module-info** (simplest; avoids JPMS complexities)

3.2 Add Source Folders

Eclipse by default gives you:

src/

Add another folder:

test/

Right-click → Build Path → Use as Source Folder.

3.3 Add External Libraries

All dependencies must be downloaded manually and placed into:

/lib

Then in Eclipse:

- Right-click project → Properties → Java Build Path → Libraries → Add JARs → select the JARs in /lib.

Dependencies needed:

- **JUnit 5 standalone**
- **Mockito** (optional)
- **CSV library** (OpenCSV or Commons CSV)
- **JFreeChart** (optional visualization)
- **SLF4J + slf4j-simple**

3.4 JUnit 5 Setup

Use:

junit-platform-console-standalone-x.x.x.jar

Add this JAR to the build path, *AND* mark it as a **Test Library**:

- Right-click → Build Path → Configure Build Path → Libraries → Add External JARs → select junit standalone.

Eclipse will then allow:

- Run → Run As → JUnit Test

3.5 Javadoc

Use:

Project → Generate Javadoc…

Eclipse auto-detects the source folder and generates HTML docs in /docs/javadoc.

3.6 Runnable JAR Export

When the application is ready:

File → Export → Runnable JAR file

Launch configuration: CreditCardSimulator (Main class)

Library handling: Package required libraries into generated JAR

Destination: /CreditCardSimulator.jar

---

4 — Updated Build-Related Design Decisions

4.1 No dependency management tool

All dependency versioning and addition is manual:

- Developer is responsible for adding updated JARs to /lib/.

4.2 No automated test runner

- JUnit tests executed within Eclipse manually.
- Continuous integration optional but outside Eclipse.

4.3 No automatic resource copying

- Eclipse does not automatically include /resources/ in the JAR.
- The build instructions should explicitly copy needed resources if required.

(We will reference resource files via relative paths or copy them to the working directory.)

4.4 No shaded JAR creation

Eclipse runnable JAR export creates a "fat JAR" automatically by bundling dependent JARs when exporting.

---

5 — Architectural Updates (small but important)

The core MVC-and-strategy architecture remains the same.

However, to accommodate Eclipse:

5.1 Replace Maven/Gradle config classes with simple Java config files

Previously:

/config/application.properties

Now:

/src/com.creditcardsim/config/DefaultConfig.java

Example:

```
public final class DefaultConfig {
    public static final BigDecimal APR = new BigDecimal("0.35");
    public static final BigDecimal PENALTY_APR = new BigDecimal("0.3999");
    public static final int BILLING_CYCLE_DAYS = 30;
    public static final int DUE_DAY = 20;

    public static final BigDecimal REWARD_GROCERIES = new BigDecimal("0.03");
    public static final BigDecimal REWARD_GAS = new BigDecimal("0.02");
    public static final BigDecimal REWARD_OTHER = new BigDecimal("0.01");

    private DefaultConfig() {}
}
```

Or load from a simple .properties file using Properties.

5.2 Logging simplified

SLF4J + slf4j-simple is easiest, but Eclipse can run without logs too.

5.3 UML diagrams generated manually

PlantUML can be used, but Eclipse does not require it.

---

6 — Updated Testing Plan

Tests remain identical, except:

6.1 Test folder uses Eclipse's test source folder

/test/com.creditcardsim.model.*

6.2 Adding JUnit 5 to run tests

JUnit tests must be run through Eclipse's built-in JUnit runner.

6.3 External test resources

Place synthetic CSVs inside:

/test/resources/

And load via:

Paths.get("test/resources/small-sample.csv")

---

7 — Updated Documentation Workflow

- UML diagrams stored in /docs/diagrams/
- Javadoc: Export manually to /docs/javadoc
- README: Updated instructions referencing Eclipse build steps

README will contain:

Installation:

1. Install Eclipse Java EE or Java Developers edition

2. Import project as "Existing Project"

3. Add JAR dependencies from /lib/

4. Run CreditCardSimulator.java

---

8 — Updated Packaging & Distribution Strategy

Since no Maven/Gradle:

Use:

File → Export → Runnable JAR

Options:

- **"Package required libraries into generated JAR"**
  → produces a fat standalone JAR that runs on macOS and Windows.

Java execution:

java -jar CreditCardSimulator.jar

---

9 — Final Eclipse-Compatible Project Layout (complete)

CreditCardSimulator/

```
├── .project
├── .classpath
├── src/
│   ├── com/creditcardsim/app/Main.java
│   ├── com/creditcardsim/controller/...
│   ├── com/creditcardsim/model/account/...
│   ├── com/creditcardsim/model/strategy/...
│   ├── com/creditcardsim/view/...
│   ├── com/creditcardsim/util/...
│   └── com/creditcardsim/config/DefaultConfig.java
├── test/
│   └── com/creditcardsim/tests/...
├── lib/
│   ├── junit-platform-console-standalone-1.x.x.jar
```

```
│   ├── mockito-core-x.x.x.jar
│   ├── jfreechart-x.x.x.jar
│   ├── commons-csv-x.x.x.jar
│   └── slf4j-simple-x.x.x.jar
├── resources/
│   ├── Carl2025.csv
│   └── default.properties (optional)
├── docs/
│   ├── diagrams/
│   └── javadoc/
├── CreditCardSimulator.launch (optional)
└── README.md
```

---

10 — What remains unchanged

Everything domain-specific remains **exactly the same** as the prior design:

- MVC structure
- Model classes (Transaction, Account, Statement, Rewards, etc.)
- Strategy pattern for Payment & Interest algorithms
- Iterator pattern
- Daily Synchrony interest algorithm
- Heavy/Lite Revolver cycles
- Minimum payment formula & hidden Synchrony constraints
- Swing UI design
- Visualization windows
- Summary statistics and CSV loading
- Use Case requirements from provided PDFs

No behavioral changes — **only build & environment changes**.

Project: CreditCardCalculatorBrionBlais Design 1.2 — Renamed Project

Java 17 • Swing/AWT • Eclipse Build • MVC • Strategy Pattern • Iterator Pattern

Payment Strategies • Interest Algorithms • Rewards • Synchrony Agreement Rules

Reads Transaction CSV • Cross-platform (macOS + Windows)

---

1 — Overview

CreditCardCalculatorBrionBlais is a desktop Java application that simulates credit card finance behavior (interest, fees, rewards, payments) based on:

- **Synchrony Bank Credit Card Account Agreement** (interest, minimum payment, fees, penalty APR)
- **Use Case Description** (UI workflow, CSV input, variations, visualizations)
- **Carl2025.csv** (actual purchase/payment data for the example scenario)

The application is based on a strict **Model-View-Controller (MVC)** architecture with:

- **Strategy Pattern** for:
  - Payment behaviors
  - Interest algorithms
- **Iterator Pattern** for transactions
- **Swing/AWT desktop UI**
- **JUnit 5 test framework**
- **Fully Eclipse-based build**, no Maven or Gradle

Runs on:

- macOS
- Windows PC/laptop

---

2 — Eclipse Project Structure

CreditCardCalculatorBrionBlais/

```
├── .project
├── .classpath
├── src/                                (main source)
│   ├── com/creditcardcalculatorbrionblais/app/Main.java
│   ├── com/creditcardcalculatorbrionblais/controller/...
│   ├── com/creditcardcalculatorbrionblais/model/account/...
│   ├── com/creditcardcalculatorbrionblais/model/transaction/...
│   ├── com/creditcardcalculatorbrionblais/model/strategy/...
│   ├── com/creditcardcalculatorbrionblais/model/iterator/...
│   ├── com/creditcardcalculatorbrionblais/view/...
│   ├── com/creditcardcalculatorbrionblais/util/...
│   └── com/creditcardcalculatorbrionblais/config/DefaultConfig.java
├── test/                               (JUnit 5 tests)
│   └── com/creditcardcalculatorbrionblais/tests/...
├── lib/                                (MANUALLY added JARs)
│   ├── junit-platform-console-standalone-x.x.x.jar
│   ├── mockito-core-x.x.x.jar
│   ├── jfreechart-x.x.x.jar
│   ├── commons-csv-x.x.x.jar or opencsv.jar
│   ├── slf4j-api.jar
│   └── slf4j-simple.jar
├── resources/
│   ├── Carl2025.csv
│   └── optional: default.properties
├── docs/
```

```
│   ├── diagrams/*.puml
│   └── javadoc/
├── CreditCardCalculatorBrionBlais.launch   (Optional Eclipse run config)
└── README.md
```

---

3 — Architectural Overview

3.1 Model–View–Controller

Model

Contains *all business logic*:

- Account
- Transaction
- Statements
- Rewards
- Interest calculation
- Fees
- Iterator for transactions
- Strategy pattern implementations

Controller

Coordinates user actions and delegates to Model:

- Loads CSV
- Validates input
- Applies strategies
- Generates Statements
- Prepares data for views
- Handles errors & warnings
- Initializes visualization windows

View

Swing/AWT user interface:

- Main window
- File selection dialog
- Parameters panel
- Summary window
- Visualization window (JFreeChart or custom)
- Error dialog

Views contain **no business logic**.

---

4 — Domain Model

4.1 Transaction

Immutable, represents a single purchase or payment.

```
public final class Transaction {
    public enum Category { GROCERIES, GAS, OTHER, PAYMENT }

    private final LocalDate date;
    private final Category category;
    private final BigDecimal amount;   // positive for purchase, negative for payment

    private final boolean isPayment;   // derived from category or sign
}
```

Validation:

- amount > 0 for purchases
- amount < 0 for payments
- category ∈ {Groceries, Gas, Other, Payment}

4.2 Account

Core financial engine.

Responsibilities:

- Maintain ordered list of transactions
- Load data from CSV
- Generate monthly statements
- Compute interest and fees
- Calculate minimum payments
- Compute rewards
- Track penalty APR state
- Provide iterator over transactions

Internal data:

- startingBalance
- APR
- penaltyAPR
- dailyRate
- cycleStartDay
- billingCycleDays
- dueDate
- minInterestCharge ($2)
- paperStatementFee ($1.99)
- list

Exposes:

- applyPaymentStrategy()
- calculateStatement()
- computeRewards()
- iterator()

4.3 Statement

Immutable snapshot for one month:

- openingBalance
- newCharges
- payments
- interest
- fees
- rewards
- minimumPayment
- closingBalance
- YearMonth period

Used for UI, CSV export, and visualization.

---

5 — Strategy Patterns

5.1 PaymentStrategy

Interface:

```
public interface PaymentStrategy {
    List<Transaction> generatePayments(AccountSnapshot snapshot, Statement statement);
}
```

Implementations:

Early Transactor

- Pays full statement balance **one day after cycle ends**
- Starting balance = 0
- Uses rewards only for gift cards (not statement credit)

Wall Street Transactor

- Pays full statement balance **on due date** (end of grace period)
- Uses rewards each month as **statement credit**
- Starting balance = 0

Light Revolver

- For 5 months: pays **minimum payment**
- 6th month: pays **full statement balance**
- Cycles
- Payments always on due date
- Starting balance configurable (default $1000)
- Rewards → gift cards

Heavy Revolver

- Pays **minimum payment** for 5 months
- 6th month: pays **5 days late**
    - Causes **late fee**
    - Advances penalty APR state
- Starting balance configurable (default $5000)
- Rewards → gift cards

5.2 InterestCalculationStrategy

```
public interface InterestCalculationStrategy {
    InterestResult computeInterest(
        AccountSnapshot snapshot,
        LocalDate cycleStart,
        LocalDate cycleEnd
    );
}
```

Implementations:

AverageDailyBalanceStrategy

Standard credit-card average daily balance computation.

SynchronyDailyBalanceStrategy *(default)*

Implements Synchrony's rules in the uploaded PDF:

- Daily balance method
- Daily compounding
- Minimum interest charge = $2 if interest > 0
- Treats late fees, returned payment fees, paper statement fees as "new purchases"
- Penalty APR logic

---

6 — Iterator Pattern

Account implements Iterable<Transaction>

TransactionIterator

- Iterates in chronological order
- Allows simple for-each interaction
- Used by interest calculator and controller

---

7 — Key Financial Rules (from Synchrony PDF)

All derived from the uploaded Synchrony agreement.

APRs:

- APR for purchases: **34.99%**
- Penalty APR: **39.99%**
- Daily rate = APR / 365

Minimum Interest Charge:

- If interest > 0 but < $2 → charge **$2**

Minimum Payment Calculation:

- Greater of:
    1. $30
    2. 3.5% of new balance
    3. (1% of new balance + interest + late fees)
- Plus past due

- Rounded up
- Never exceeds new balance

Fees:

- Late Payment Fee:
    - $30 or $41 depending on last 6 cycles
- Returned Payment Fee: same tiered system
- Paper Statement Fee: $1.99 if paper statements selected

Penalty APR:

- Triggered by **2 late payments within 12 cycles**
- May remain indefinitely
- Applies to all transactions once active

---

8 — Daily Interest Algorithm (Synchrony)

Day-by-day within cycle:

```
balanceDayStart = previousBalance

for each day:
    balanceDayStart += new purchases
    balanceDayStart -= payments
    balanceDayStart += fees (treated like new purchases)

    if balanceDayStart > 0:
        dailyInterest = balanceDayStart * dailyRate
        balanceDayEnd = balanceDayStart + dailyInterest
    else:
        dailyInterest = 0
        balanceDayEnd = balanceDayStart
```

accumulate dailyInterest

At cycle end:

- If totalInterest < $2 → use $2
- Add interest to closing balance

---

9 — Controller Layer

Main controller responsibilities:

1. Start application
2. Load CSV
3. Validate date/category/amount
4. Configure strategy objects
5. Run simulation cycle-by-cycle
6. Produce Statements
7. Provide data to UI
8. Open visualizations
9. Error & exception handling

Controllers include:

- **MainWindowController**
- **FileLoadController**
- **SimulationController**
- **VisualizationController**

Each controller interacts with model but never does business logic itself.

---

10 — View Layer (Swing/AWT)

Main Window (JFrame)

Contains:

- File selection
- Strategy selection dropdown
- Parameter configuration panel
- APR / starting balance override fields
- Starting date / ending date fields
- Rewards percentages (optional)
- Simulate button

Summary Window

- Displays:
    - Opening balance
    - Ending balance
    - Total interest
    - Total fees
    - Total rewards
    - Total payments
- Export button (CSV)

Visualization Window

Tabs:

- Daily
- Monthly
- Yearly

Charts:

- Balance vs Time
- Interest over Time
- Fees over Time
- Payment events

Swing tools:

- Panels
- JTable
- JScrollPane
- Graphics2D or JFreeChart

---

11 — CSV Handling

CSV contains:

yyyy-mm-dd, category, amount

Categories:

- Groceries
- Gas
- Other

All purchases assumed on the 15th per your scenario, but CSV may override this.

Loader responsibilities:

- Check formatting
- Ensure valid dates
- Ensure valid categories
- Parse BigDecimal safely
- Produce Transaction objects
- Sort chronologically

---

12 — Testing Plan (JUnit 5)

Unit Tests

- TransactionTest
- CsvLoaderTest
- AccountInterestTest
- MinimumPaymentTest

- PenaltyAPRTest
- Payment strategy tests
- Interest strategy tests

Integration Tests

- End-to-end simulation using **Carl2025.csv**
- Expected totals for:
    - interest
    - rewards
    - fees
    - ending balance

Mocking

Using Mockito for:

- PaymentStrategy mocks
- InterestCalculationStrategy mocks
- Controller behavior tests

---

13 — Logging

SLF4J + slf4j-simple:

- INFO for normal operations
- WARN for odd but recoverable cases
- ERROR for parsing or unexpected problems

---

14 — JAR Packaging

Using Eclipse:

File → Export → Runnable JAR

→ Launch configuration: Main class

→ Package required libraries into generated JAR

→ Output: CreditCardCalculatorBrionBlais.jar

This JAR will work on macOS and Windows with:

java -jar CreditCardCalculatorBrionBlais.jar

---

15 — Documentation

Included in project:

- /docs/Architecture-CreditCardCalculatorBrionBlais.md
- /docs/diagrams/*.puml
- /docs/javadoc/
- /README.md

Diagrams:

1. Class diagram
2. Sequence diagram (billing cycle)
3. Package diagram
4. State machine diagram (Penalty APR lifecycle)

---

16 — Development Milestones

1. **Project skeleton** (Eclipse project, packages, stubs)
2. **Core Model** (Transaction, Account, CSV Loader)
3. **Interest Algorithms + Tests**
4. **Payment Strategies + Tests**
5. **Controllers**
6. **Basic Swing UI**
7. **Visualizations**
8. **Documentation & JAR Export**

---

17 — Summary

The **CreditCardCalculatorBrionBlais** design is now fully specified:

- Eclipse-based Java 17 project
- MVC architecture
- Strategy & Iterator patterns
- Swing UI
- Synchrony rules implemented precisely
- CSV-driven domain model
- Four payment strategies
- Daily compounding interest
- Penalty APR
- JUnit 5 test suite
- Runnable standalone JAR

This design is stable and ready for implementation.

Would you like the next step?

I can now generate:

A. Full project skeleton (all Java files, empty methods, ready to import into Eclipse)

B. Full working model layer (Account, Transaction, Strategies, InterestCalc)

C. Full working project including UI

D. UML diagrams (PlantUML)

E. A PDF-style design document

Just tell me which you want.

Final Reflections

- There were very few changes required to the design, as the prompts were very detailed
- Generative AI saves an incredible amount of time when you consider the amount of time needed to draft the equivalent amount of information.
- Without sufficient practical experience, it would be difficult to determine if the design is appropriate and technically feasible, without implementing it and testing. The scope of the design is so substantial that implementing it without Generative AI support would be very difficult or time consuming.

Implementation

I created a complete **Eclipse-ready project** for **CreditCardCalculatorBrionBlais** (model, strategies, a simple Synchrony-style interest calculator, CSV loader, controller, and a minimal Swing UI), plus a basic JUnit 5 test.

Download the project ZIP here:
Download CreditCardCalculatorBrionBlais_project.zip

What I included

- src/ full Java source under com.creditcardcalculatorbrionblais:
    - Model: Transaction, Account, SimulationResult
    - Strategies: PaymentStrategy + EarlyTransactorStrategy, WallStreetTransactorStrategy, InterestCalculationStrategy, SynchronyDailyBalanceStrategy
    - CSV loader: CsvLoader
    - Controller: MainWindowController
    - View: MainWindow (Swing UI)
    - App entry: Main
    - Config: DefaultConfig
- resources/Carl2025.csv — sample transactions
- test/ with a basic JUnit 5 smoke test
- README.md with import/run hints

How to run

1. Unzip and import into **Eclipse**: File → Import → Existing Projects into Workspace (or import as General → Existing Projects).
2. Add any required JARs to /lib/ to the project's Build Path if you want unit tests (JUnit 5). The app itself uses only the JDK.
3. Run com.creditcardcalculatorbrionblais.app.Main as a Java Application.
4. In the UI: Load resources/Carl2025.csv and click **Simulate**. Choose strategy (Early or WallStreet).

Notes & next steps

- This is a **functional starting point**, intentionally minimal but working. It implements the main architecture, a Synchrony-like interest algorithm, and two payment strategies.
- I can:
    - Expand the payment strategies (LightRevolver, HeavyRevolver) with full behavior and tests.
    - Implement full minimum payment formula and penalty APR lifecycle tracking.
    - Add detailed visualizations (JFreeChart), CSV export, and full Javadoc.
    - Harden parsing and edge cases, and add more unit/integration tests.
    - Create an Eclipse .project and .launch preset if you want.

Which enhancement should I implement next?

Final Thoughts and Reflections

- AI systems used:
  - Google AI – multiple mistakes within the code and was difficult to use
  - ChatGPT – was able to use a complex prompt and it produced a large codebase; however, the free service was quickly exhausted
  - Claude – comprehensive commenting and code refactoring, but quickly exhausted free services
  - Microsoft Copilot – workhorse of AI, was able to use it to do Javadoc comments, method writing and refactoring. However, it had issues maintaining the file structures and caused multiple problems when trying to update the GUI.
- This project was difficult based on the size of the codebase needed and the limited time. I spent more time trying to get the AI systems to work consistently than I did with actual code review.
- At this time, I am inclined to use the AI systems to generate ideas for particular methods, Javadoc commenting and suggesting for broad structure. I do not have confidence in the systems ability to produce a program that meets the full specifications. The amount of time required to refine what is produced and fix bugs would be better spent actually coding.