Name: Samer Amri

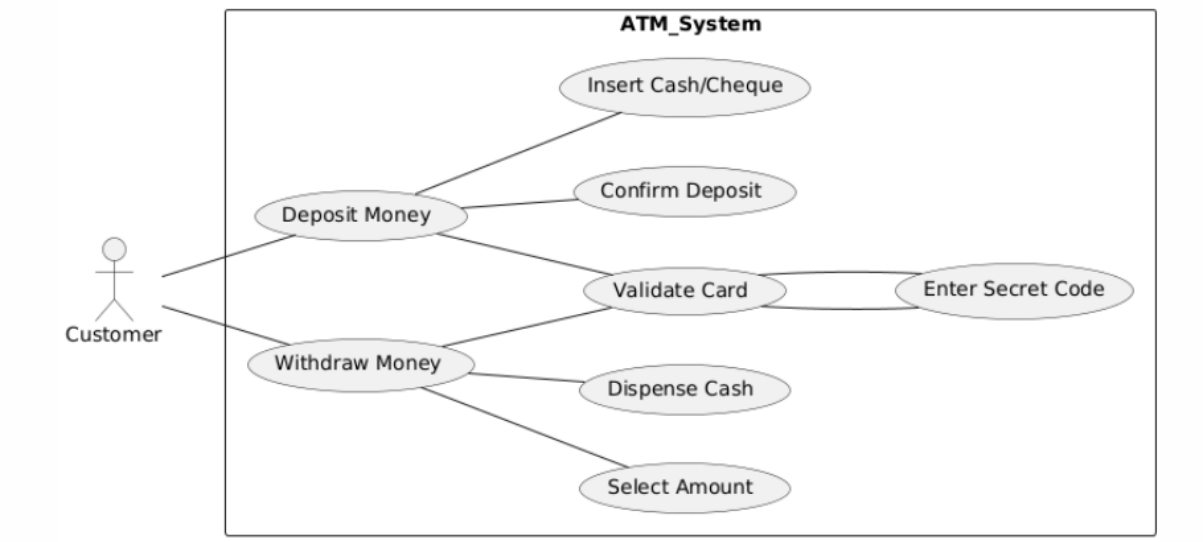
M2: IAGE

Hands-out N° 1: Object-oriented approach vs Process-oriented approach

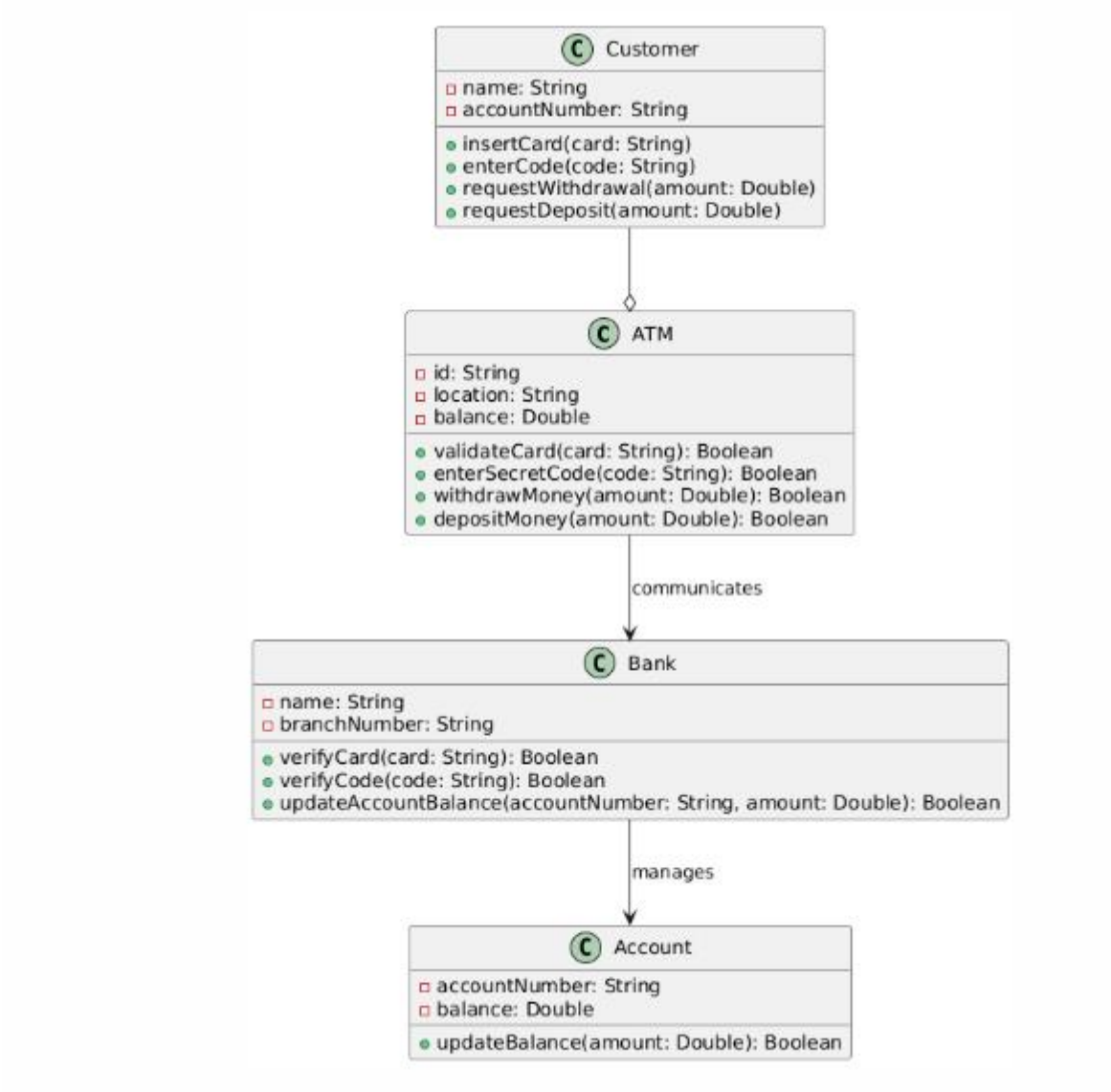
Objective 1:

Task 1 :

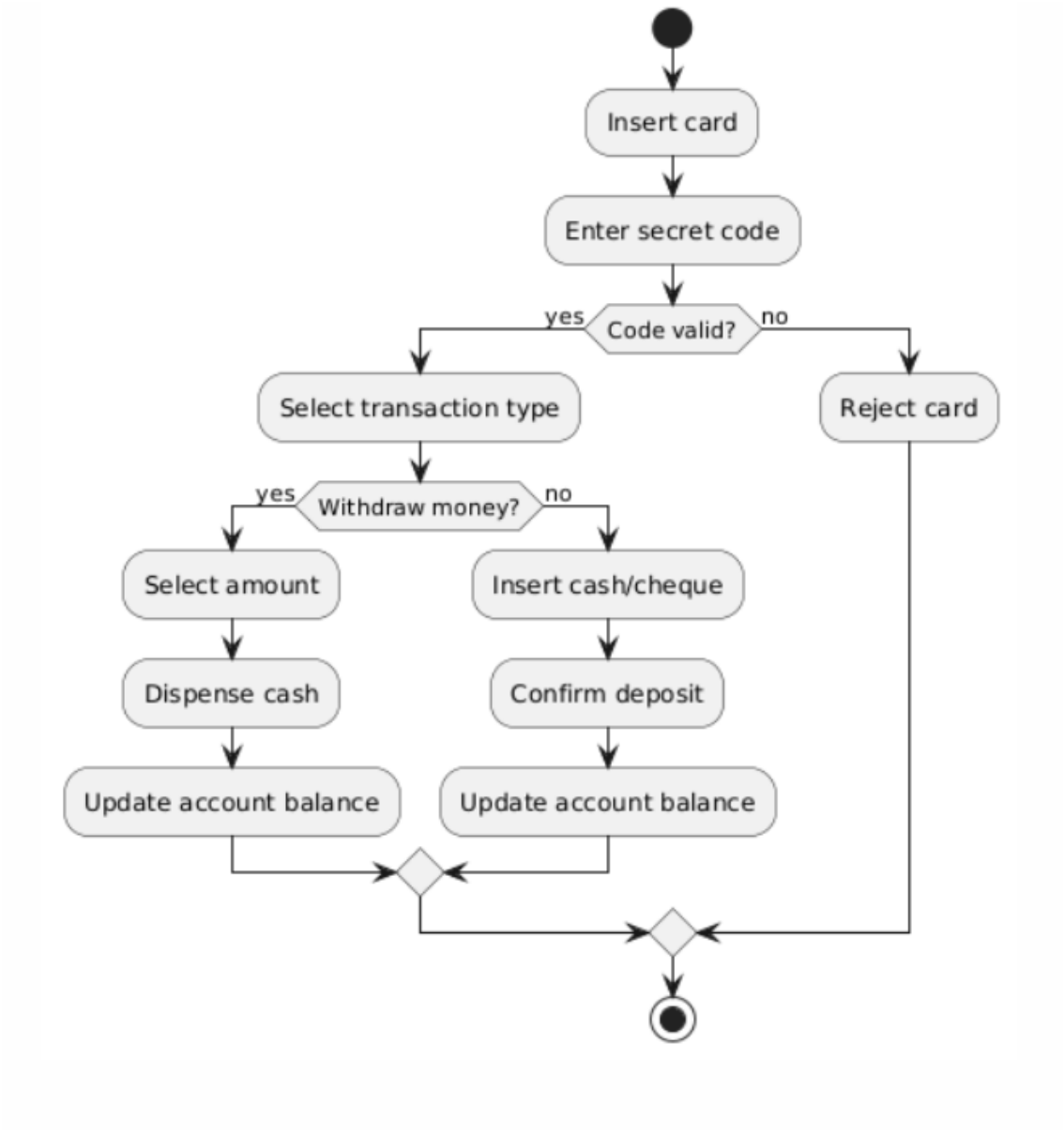
Use Case diagram :



Class Diagram :

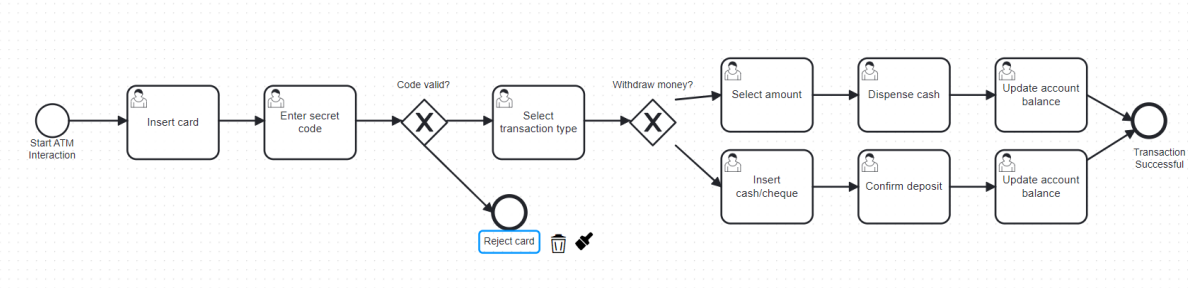


Activity Diagram:



Task 2:

ATM using camunda :



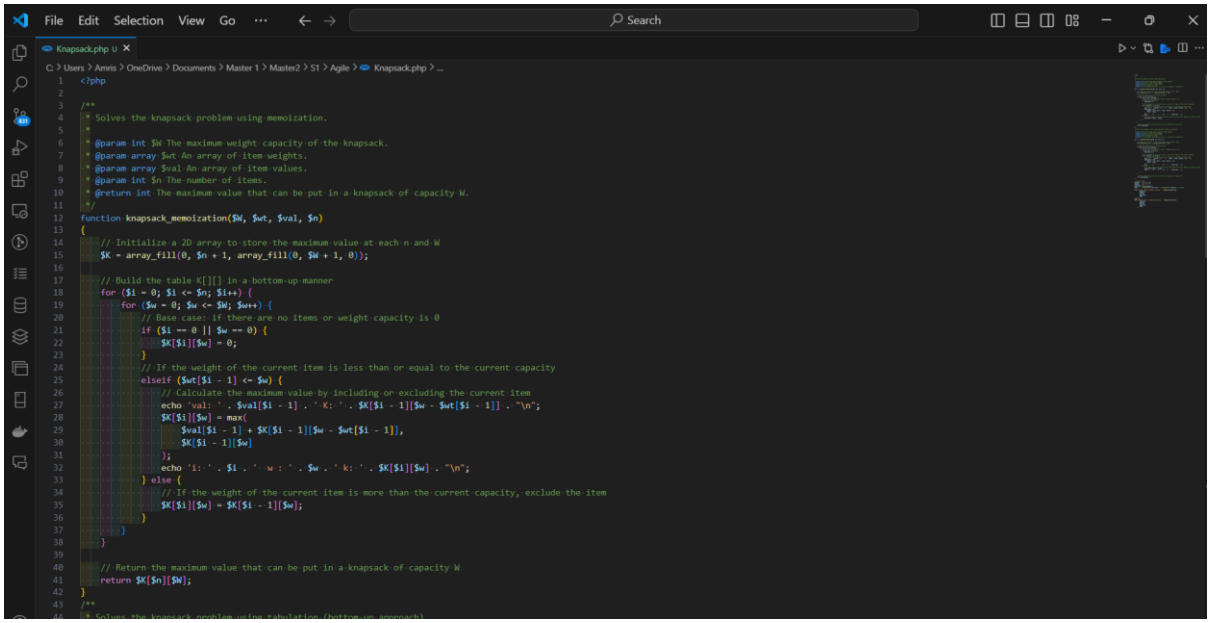
Category	Object-Oriented Approach	Process-Oriented Approach

Actors	A*ctors* are the entities interacting with the system. Here, the primary actors are the **Customer** (cardholder) and **ATM**. The customer interacts with the ATM using their card and PIN, while the ATM provides access to the customer's bank account and handles transactions like withdrawals and deposits.	The focus is on the roles and participants in each process. The **Customer** initiates processes like "Withdraw Money" or "Deposit Money" by interacting with the ATM. The ATM is not seen as an object but as a part of the overall process that guides the transaction flow. For example, the customer interacts with the "Verify PIN" process to proceed with other transactions.
Use Cases	Represent the functionalities that the actors can perform. The primary use cases include: **"Withdraw Money"** and **"Deposit Money"**. Each use case defines a specific interaction between the **Customer** and the **ATM**. For example, the **"Withdraw Money"** use case involves the customer inserting their card, entering the PIN, selecting an amount, and receiving cash. These use cases are typically documented through use case diagrams or descriptions.	Are often framed as the sequence of steps required to complete a business process. For instance, the **"Withdraw Money"** process includes steps like "Insert Card," "Enter PIN," "Select Withdrawal Amount," "Verify Balance," and "Dispense Cash." Each step is treated as a subprocess or activity within the overall transaction. This approach often uses flowcharts or BPMN (Business Process Model and Notation) diagrams.
Static Behaviors	emphasizes *static behaviors* by modeling the structural aspects of the ATM system, such as classes and their relationships.	The process-oriented approach focuses on defining the state and workflow of each process. It details the static structure of processes,

	For example, classes like **ATM**, **Customer**, **Account**, **Transaction**, and **Card** could be defined. Each class encapsulates attributes and methods relevant to the entity. The **ATM** class might have methods like `authenticateUser()` and `processTransaction()`, while the **Transaction** class could have attributes like `amount` and `type` (withdrawal or deposit). UML class diagrams are often used to represent these static behaviors.	such as the various states the ATM process can be in (e.g., "Idle," "Card Inserted," "PIN Verified," "Transaction in Progress"). Each state has conditions that determine transitions, like moving from "PIN Verified" to "Select Transaction." These can be represented using state diagrams or flowcharts that show the fixed sequence of operations and how the system moves between states.
Dynamic Behaviors	Describe how objects interact with each other over time to accomplish a particular use case. This could involve **sequence diagrams** that depict the interactions between the **Customer**, **ATM**, **Account**, and **Transaction** objects. For example, a sequence diagram for a **"Withdraw Money"** use case would show the customer entering their card, the ATM validating the PIN, querying the account balance, and dispensing cash if sufficient funds are available.	Focus on the flow of activities and the transitions between different steps in a process. For example, a **"Withdraw Money"** process might start with the customer inserting the card and then proceed through a series of steps: "Enter PIN," "Check Account Balance," "Dispense Cash," and "Print Receipt." Each step transitions smoothly to the next based on conditions, such as a successful PIN verification. This approach uses activity diagrams or BPMN diagrams to represent the flow of events and decision points in the process.

Objective 2: Dynamic Programing (DP) :

Knapsack problem :



```php
<?php

/**
 * Solves the knapsack problem using memoization.
 *
 * @param int $W The maximum weight capacity of the knapsack.
 * @param array $wt An array of item weights.
 * @param array $val An array of item values.
 * @param int $n The number of items.
 * @return int The maximum value that can be put in a knapsack of capacity W.
 */
function knapsack_memoization($W, $wt, $val, $n)
{
    // Initialize a 2D array to store the maximum value at each n and W
    $K = array_fill(0, $n + 1, array_fill(0, $W + 1, 0));

    // Build the table K[][] in a bottom-up manner
    for ($i = 0; $i <= $n; $i++) {
        for ($w = 0; $w <= $W; $w++) {
            // Base case: if there are no items or weight capacity is 0
            if ($i == 0 || $w == 0) {
                $K[$i][$w] = 0;
            }
            // If the weight of the current item is less than or equal to the current capacity
            elseif ($wt[$i - 1] <= $w) {
                // Calculate the maximum value by including or excluding the current item
                echo 'val: ' . $val[$i - 1] . ' K: ' . $K[$i - 1][$w - $wt[$i - 1]] . "\n";
                $K[$i][$w] = max(
                    $val[$i - 1] + $K[$i - 1][$w - $wt[$i - 1]],
                    $K[$i - 1][$w]
                );
                echo 'i: ' . $i . '  w : ' . $w . ' k: ' . $K[$i][$w] . "\n";
            } else {
                // If the weight of the current item is more than the current capacity, exclude the item
                $K[$i][$w] = $K[$i - 1][$w];
            }
        }
    }

    // Return the maximum value that can be put in a knapsack of capacity W
    return $K[$n][$W];
}
/**
 * Solves the knapsack problem using tabulation (bottom-up approach).
 *
 * @param int $W The maximum weight capacity of the knapsack.
 * @param array $wt An array of item weights.
 * @param array $val An array of item values.
 * @param int $n The number of items.
 * @return int The maximum value that can be put in a knapsack of capacity W.
 */
function knapsack_tabulation($W, $wt, $val, $n)
{
    // Initialize a 2D array to store the maximum value at each n and W
    $K = array_fill(0, $n + 1, array_fill(0, $W + 1, 0));

    // Build the table K[][] in a bottom-up manner
    for ($i = 0; $i <= $n; $i++) {
        for ($w = 0; $w <= $W; $w++) {
            // Base case: if there are no items or weight capacity is 0
            if ($i == 0 || $w == 0) {
                $K[$i][$w] = 0;
            }
            // If the weight of the current item is less than or equal to the current capacity
            elseif ($wt[$i - 1] <= $w) {
                // Calculate the maximum value by including or excluding the current item
                echo 'val: ' . $val[$i - 1] . ' K: ' . $K[$i - 1][$w - $wt[$i - 1]] . "\n";
                $K[$i][$w] = max(
                    $val[$i - 1] + $K[$i - 1][$w - $wt[$i - 1]],
                    $K[$i - 1][$w]
                );
                echo 'i: ' . $i . '  w : ' . $w . ' k: ' . $K[$i][$w] . "\n";
            } else {
                // If the weight of the current item is more than the current capacity, exclude the item
                $K[$i][$w] = $K[$i - 1][$w];
                echo 'i: ' . $i . '  w : ' . $w . ' k: ' . $K[$i][$w] . "\n";
            }
        }
    }

    // Return the maximum value that can be put in a knapsack of capacity W
    return $K[$n][$W];
}

// Example usage
$weights  = [1, 3, 4, 5];
```

```php
function knapsack_tabulation($W, $wt, $val, $n)
        }
    }

    // Return the maximum value that can be put in a knapsack of capacity W
    return $K[$n][$W];
}

// Example usage
$weights  = [1, 3, 4, 5];
$values   = [1, 4, 5, 7];
$capacity = 7;
$num_items = count($weights);
$memo     = array_fill(0, $num_items + 1, array_fill(0, $capacity + 1, null));

echo "Maximum value in knapsack top down: " . knapsack_memoization(
    $capacity,
    $weights,
    $values,
    $num_items,
    $memo
);
echo "\n";
echo "Maximum value in knapsack bottom up: " . knapsack_tabulation(
    $capacity,
    $weights,
    $values,
    $num_items,
    $memo
);
```

Discussion about Top-Down (Memoization) and Bottom-Up (Tabulation) Approaches :

Speed and Efficiency :

1. **Top-Down (Memoization)**:
   - **Speed**: Slower because it uses recursive calls.
   - **Efficiency**: Uses recursion and stores results to avoid repeating calculations.
   - **Time Complexity**: $O(n \cdot W)$
   - **Space Complexity**: $O(n \cdot W)$ because of the memoization table.

2. **Bottom-Up (Tabulation)**:
   - **Speed**: Faster because it avoids recursive calls.
   - **Efficiency**: Uses loops to build the solution step-by-step.
   - **Time Complexity**: $O(n \cdot W)$
   - **Space Complexity**: $O(n \cdot W)$ because of the DP table.

Use Cases

1. **Top-Down (Memoization)**:
   - Good for problems that can be broken into smaller, overlapping subproblems.
   - Easier to write if you are comfortable with recursion.
   - Best for small to medium-sized problems.

2. **Bottom-Up (Tabulation)**:
   - Good for problems that can be solved iteratively.
   - Better for larger problems because it is faster.
   - Best when the problem can be represented in a table.

Conclusion

- **Best Approach**: Bottom-up (tabulation) is usually better for larger problems because it is faster and avoids the overhead of recursive calls.

- **When to Use Top-Down**: Use top-down (memoization) for smaller problems or when recursion is more natural.

Dynamic Programming

Dynamic programming is a technique used to solve problems by breaking them down into simpler subproblems and storing the results. This avoids redundant calculations and makes the solution more efficient. It is especially useful for optimization problems like the knapsack problem, where it helps to find the best solution in a reasonable amount of time.