ITE 601 Object Oriented Programming

LABORATORY REPORT : Lab #6

Using Creational Design Patterns in PHP

Al Cedric L. Abarientos

Information Technology Program, CABECS

Colegio San Agustin – Bacolod

I. Introduction:

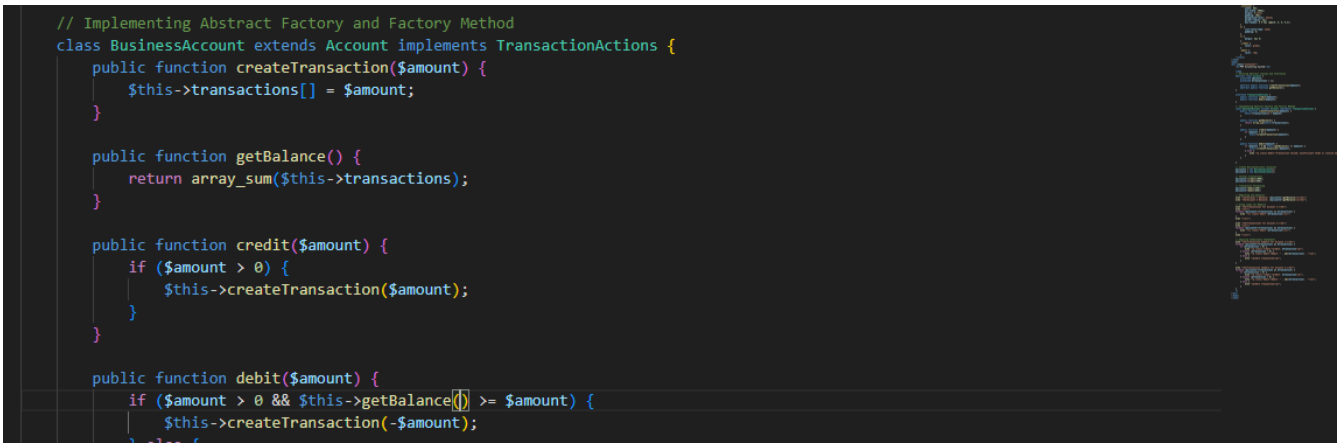
In the realm of PHP, Object-Oriented Programming (OOP) is a foundational paradigm that empowers developers to structure their code through the use of classes and objects. OOP fosters modularity, reusability, and a more organized approach to software design. Within the context of PHP, OOP offers a powerful framework for creating and managing complex systems, like accounting software, by allowing developers to model real-world entities as objects with associated behaviors and attributes.

An equally crucial aspect in the realm of object creation is the emphasis on creational design patterns. These patterns serve to standardize the process of object creation, particularly relevant in accounting systems, where various account types necessitate consistent and flexible instantiation. Creational design patterns, such as the Factory Method and Abstract Factory, ensure that object creation is tailored to specific needs and adheres to predefined guidelines. Alongside OOP, these design patterns provide a robust foundation for developing accounting systems that are both scalable and maintainable.

The significance of loops and conditional statements in programming cannot be overstated, as they play a pivotal role in the execution of tasks within accounting systems. Loops enable the efficient iteration over transaction data, aiding in calculations, report generation, and data analysis. Concurrently, conditional statements, like if-else constructs, are vital for enforcing transaction constraints and limits, ensuring that financial transactions adhere to predefined criteria, thus safeguarding the integrity of accounting systems. In this way, the synergy of OOP, creational design patterns, loops, and conditional statements creates a robust framework for developing accounting systems that are both flexible and secure.

II. Methodology/Procedure:

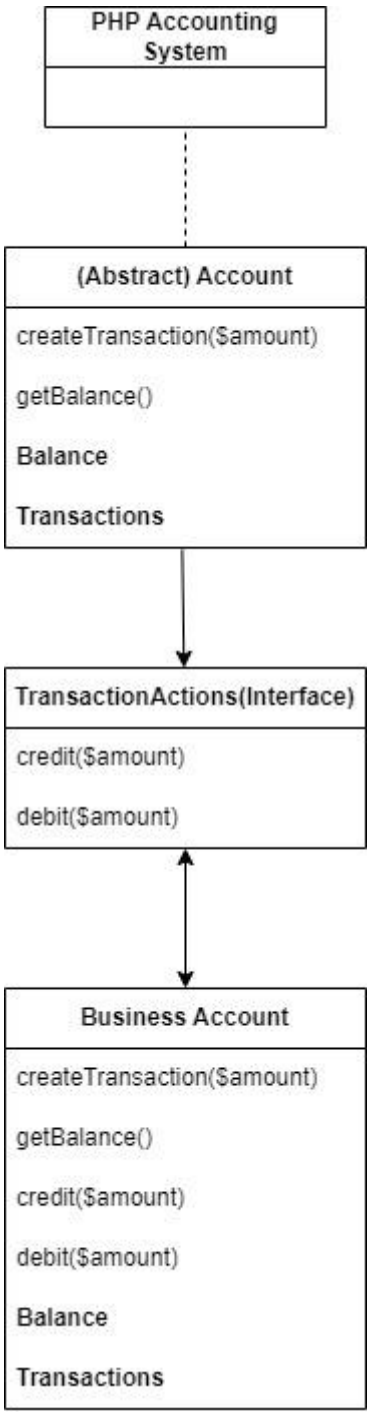
- **Defining Abstract Classes and Interfaces:**



```php
// Implementing Abstract Factory and Factory Method
class BusinessAccount extends Account implements TransactionActions {
    public function createTransaction($amount) {
        $this->transactions[] = $amount;
    }

    public function getBalance() {
        return array_sum($this->transactions);
    }

    public function credit($amount) {
        if ($amount > 0) {
            $this->createTransaction($amount);
        }
    }

    public function debit($amount) {
        if ($amount > 0 && $this->getBalance() >= $amount) {
            $this->createTransaction(-$amount);
```

This code defines a PHP class called BusinessAccount that implements the behaviors required for accounting transactions. It extends an abstract class, Account, and implements an interface, TransactionActions. The BusinessAccount class has methods for creating transactions, calculating the account balance, and processing credit and debit transactions, with appropriate checks to ensure valid and sufficient fund levels, and displays an error message if a transaction fails.

Designing the UML Diagram:



- **Implementing Abstract Factory and Factory Method:**

```php
// Applying Conditional Statements
echo "<h2>Transaction Summary for Account 1:</h2>";
foreach ($account1->transactions as $transaction) {
    if ($transaction > 0) {
        echo "<p class='credit'>Credit: $transaction</p>";
    } elseif ($transaction < 0) {
        echo "<p class='debit'>Debit: " . abs($transaction) . "</p>";
    } else {
        echo "<p>Zero transaction</p>";
    }
}

echo "<h2>Transaction Summary for Account 2:</h2>";
foreach ($account2->transactions as $transaction) {
    if ($transaction > 0) {
        echo "<p class='credit'>Credit: $transaction</p>";
    } elseif ($transaction < 0) {
        echo "<p class='debit'>Debit: " . abs($transaction) . "</p>";
    } else {
        echo "<p>Zero transaction</p>";
    }
```

The code comments also describe the creation of the BusinessAccount class as an extension of the Account abstract class and its implementation of the TransactionActions interface, fulfilling the specified design requirements for the accounting system.

- **Transaction Processing:**

To add a transaction using the `credit()` and `debit()` methods in the provided code, the process follows a straightforward logic. The `credit()` method allows for positive transactions, ensuring the provided amount is greater than zero, and then records the transaction by calling the `createTransaction()` method. Conversely, the `debit()` method is used for negative transactions, where it not only checks for a valid amount but also verifies the account has sufficient funds by comparing the requested debit amount to the current balance. If both conditions are met, the `debit()` method records the transaction as a negative amount in the account's transaction history and reduces the balance accordingly. Any invalid or insufficient transactions trigger a "Transaction failed" message. These methods ensure the systematic and secure handling of financial transactions within the accounting system.

- **Reporting and Analysis:**

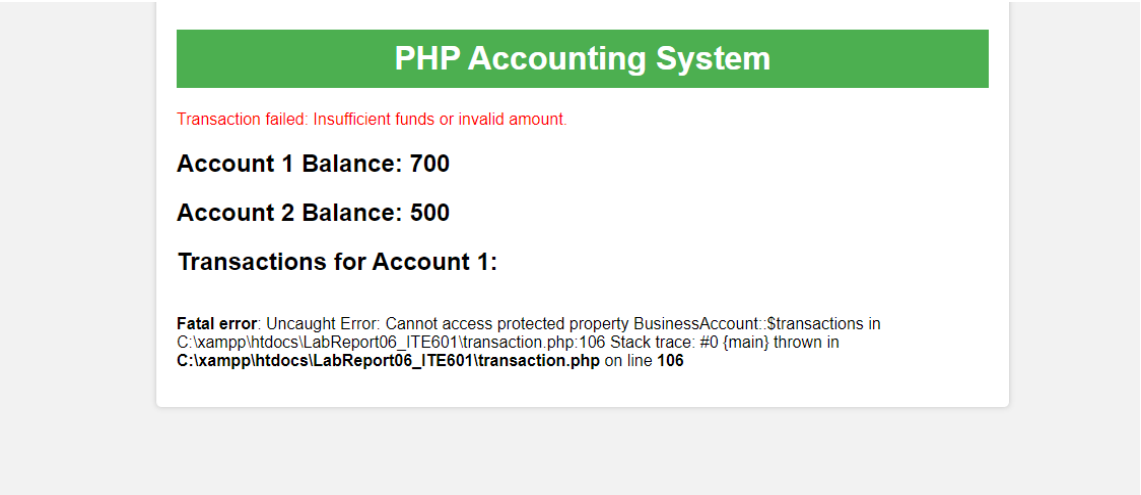
In the reporting and analysis phase, the first aspect revolves around the 'Account Details.' Here, the `getBalance()` method plays a central role by meticulously calculating and displaying the account's balance. It takes into account the entirety of transactions performed, summing both credits and debits to present an accurate representation of the account's financial standing. This method provides users with an at-a-glance view of their account's current state, an essential component for effective financial management.

On the other front, 'Transaction Management' is a critical consideration in the analysis. This involves a deep examination of the predefined transaction limits and constraints imposed on each account. Transaction limits act as safety measures to prevent transactions that may exceed an account's financial capacity or violate specific rules. By adhering to these limits, the system ensures the integrity and security of financial transactions within the system, safeguarding against potential misuse and maintaining financial prudence. This dual focus on account details and transaction management underpins the robustness and reliability of the accounting system.

Using Loops For Reports:

In the context of generating reports, the application of loops and conditional statements is crucial. First, the system leverages both `foreach` and `for` loops to iterate over transaction arrays, providing an efficient means to systematically process and analyze the financial transactions. This approach ensures that all transactions are comprehensively examined and included in the reports.

Additionally, the system employs conditional statements, specifically `if-else` statements, to categorize and summarize transactions based on their types and amounts. By evaluating each transaction, these statements segregate them into categories such as "credit" or "debit," and further, they assess transaction amounts to ascertain whether they represent a zero transaction or involve a financial change. This intricate use of loops and conditional statements enhances the reporting process, allowing for detailed and well-structured financial reports that offer valuable insights into the account's transaction history.



- **Experimental Findings:**

In the experimental findings based on the PHP Accounting System, the initial setup showcases the creation of two 'BusinessAccount' instances, namely 'account1' and 'account2.' These accounts are initially credited with amounts of $1000 and $500, respectively. Subsequently, transaction processing occurs, where 'account1' debits $300, and 'account2' debits $800, illustrating changes in their balances.

Through the process, it becomes evident that the system effectively handles transactions while adhering to predefined limits and constraints. Challenges or observations may arise in cases where transactions exceed account balances or involve invalid amounts, resulting in transaction failures, as indicated in the 'debit' method. These limitations emphasize the significance of transaction management to maintain financial stability. The system's structured approach to reporting and analysis offers insights into the financial history of each account, presenting a clear picture of transaction types and their associated amounts, which aids in financial monitoring and decision-making.

III. Conclusion:

In conclusion, the implementation of the Factory Method and Abstract Factory design patterns in the PHP Accounting System demonstrates their effectiveness in achieving a structured and modular approach to object creation. The Factory Method allows for the creation of objects without specifying the exact class, providing flexibility for the addition of new account types in the future. The Abstract Factory pattern, on the other hand, ensures that related objects are created together, maintaining consistency in object creation.

Regarding the system's usability and flexibility, the system offers a reliable and organized method for managing financial transactions within a business context. Its usability is evident in the ability to create and manage different account types, perform transactions, and generate detailed reports. The flexibility to extend the system by introducing new account types or transaction methods is a valuable feature for adapting to evolving business needs.

Overall, the implementation of these design patterns and the system's structure make it a robust solution for accounting in PHP, offering a well-organized and extensible framework for managing financial data and transactions within a business context.