Análisis de patrón de diseño

(Abstract Factory Pattern)

Rubén Darío Franco Salinas – 202012119

El patrón de diseño Abstract Factory es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Permite encapsular el proceso de creación de objetos y ocultar los detalles de implementación para crear instancias de una clase específica o de un grupo de clases.

La idea principal detrás de este patrón es definir una clase abstracta (o una interfaz) llamada "Fábrica Abstracta" que declare un conjunto de métodos de fábrica para crear objetos. Cada método de fábrica corresponde a un tipo específico de objeto que pertenece a una familia o grupo de objetos relacionados.

La implementación del código que analicé sirve para organizar las cuentas bancarias de una persona, o de un grupo de personas, de modo que el usuario pueda obtener información sobre las tasas de intereses de cada una de estas cuentas, para así manejar sus finanzas de una forma más organizada.

El diseño que se implementó se divide en 2 partes principales: La aplicación principal y el modelo. En el segundo, la persona que desarrolló el código creó clases para cada tipo de cuenta bancaria, separándolas, teniendo en cuenta qué gastos se suplirán con cada una. Por otro lado, también estableció cómo sería el manejo de intereses dependiendo del tipo de interés que se establezca para cada cuenta.

Uno de los inconvenientes que pudo tener el desarrollo del proyecto, que fue el que tuve con mi grupo en el primer proyecto del curso, fue el acoplamiento entre clases, lo que podría terminar en tener una baja coherencia y encapsulamiento en el código. Al haber tantas clases que hacen la parte de la creación de las cuentas, si el desarrollador no hubiera tenido cuidado, era posible que la aplicación principal fuera quien hiciera todas las operaciones en las clases de esta.

Ahora, en lo que tiene que ver al patrón de diseño, el desarrollador del código hizo varias “fábricas” para generar cada tipo de cuenta, de modo que todas estas extiendan de una “fábrica de fábricas”

(la fábrica abstracta) que será con quién se comunicará el usuario. A continuación, se muestra el código fuente de cada una de estas:

- **Fábrica de fábricas, de la que extienden las demás:**

```java
package org.trishinfotech.abstractfactory;

import org.trishinfotech.abstractfactory.acct.Account;
import org.trishinfotech.abstractfactory.acct.AccountOpeningDetails;

public abstract class AbstractAccountFactory {

	public abstract Account createAccount(AccountOpeningDetails opendingDetails);

}
```

- **Fábrica de depósitos:**

```java
package org.trishinfotech.abstractfactory.acct.deposit;

import org.trishinfotech.abstractfactory.AbstractAccountFactory;
import org.trishinfotech.abstractfactory.acct.AccountOpeningDetails;
import org.trishinfotech.abstractfactory.acct.AccountType;

public class DepositAccountFactory extends AbstractAccountFactory {

	@Override
	public DepositAccount createAccount(AccountOpeningDetails openingDetails) {
		if (openingDetails == null || openingDetails.isNotValid(true)) {
			throw new IllegalArgumentException("Account Opening Details are not valid!");
		}
		DepositAccount account = null;
		AccountType type = openingDetails.getAccountType();
		switch (type) {
		case RECURRING_DEPOSIT:
			account = new RecurringDepositAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName(), openingDetails.getTermInMonths());
			break;
		case FIXED_DEPOSIT:
			account = new FixedDepositAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName(), openingDetails.getTermInMonths());
			break;
		default:
			System.err.println("Unknown/unsupported account-type.");
		}
		return account;
	}
}
```

- **Fábrica de ahorros:**

```java
package org.trishinfotech.abstractfactory.acct.saving;

import org.trishinfotech.abstractfactory.AbstractAccountFactory;
import org.trishinfotech.abstractfactory.acct.Account;
import org.trishinfotech.abstractfactory.acct.AccountOpeningDetails;
import org.trishinfotech.abstractfactory.acct.AccountType;

public class AccountFactory extends AbstractAccountFactory {

	@Override
	public Account createAccount(AccountOpeningDetails openingDetails) {
		if (openingDetails == null || openingDetails.isNotValid(false)) {
			throw new IllegalArgumentException("Account Opening Details are not valid!");
		}
		Account account = null;
		AccountType type = openingDetails.getAccountType();
		switch (type) {
		case SAVING:
			account = new SavingAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName());
			break;
		case CURRENT:
			account = new CurrentAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName());
			break;
		default:
			System.err.println("Unknown/unsupported account-type.");
		}
		return account;
	}
}
```

- **Fábrica de préstamos:**

```java
package org.trishinfotech.abstractfactory.acct.loan;

import org.trishinfotech.abstractfactory.AbstractAccountFactory;
import org.trishinfotech.abstractfactory.acct.AccountOpeningDetails;
import org.trishinfotech.abstractfactory.acct.AccountType;

public class LoanAccountFactory extends AbstractAccountFactory {

	@Override
	public LoanAccount createAccount(AccountOpeningDetails openingDetails) {
		if (openingDetails == null || openingDetails.isNotValid(true)) {
			throw new IllegalArgumentException("Account Opening Details are not valid!");
		}
		LoanAccount account = null;
		AccountType type = openingDetails.getAccountType();
		switch (type) {
		case PERSONAL_LOAN:
			account = new PersonalLoanAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName(),
					openingDetails.getTermInMonths());
			break;
		case HOME_LOAN:
			account = new HomeLoanAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName(),
					openingDetails.getTermInMonths());
			break;
		case VEHICLE_LOAN:
			account = new VehicleLoanAccount(openingDetails.getAccountNo(), openingDetails.getAccountHolderName(),
					openingDetails.getTermInMonths());
			break;
		default:
			System.err.println("Unknown/unsupported account-type.");
		}
		return account;
	}
}
```

Cabe aclarar que cada una de estas sub-fábricas se encarga del manejo de un tipo de cuenta en particular (podría haber más de una cuenta del mismo tipo destinadas a diferentes gastos). Por ejemplo:

Préstamos.

- Préstamo vehicular.

```java
package org.trishinfotech.abstractfactory.acct.loan;

import org.trishinfotech.abstractfactory.acct.AccountType;
import org.trishinfotech.abstractfactory.acct.InterestStrategy;

public class VehicleLoanAccount extends LoanAccount {

	public VehicleLoanAccount(long accountNo, String accountHolderName, int termInMonths) {
		super(accountNo, accountHolderName, AccountType.VEHICLE_LOAN, termInMonths);
		setInterestStrategy(InterestStrategy.COMPOUND);
	}

	@Override
	public double getInterest(int term) {
		return this.getInterestStrategy().getInterest(accountType, amount, term);
	}

}
```

- Préstamo de vivienda.

```java
package org.trishinfotech.abstractfactory.acct.loan;

import org.trishinfotech.abstractfactory.acct.AccountType;
import org.trishinfotech.abstractfactory.acct.InterestStrategy;

public class HomeLoanAccount extends LoanAccount {

	public HomeLoanAccount(long accountNo, String accountHolderName, int termInMonths) {
		super(accountNo, accountHolderName, AccountType.HOME_LOAN, termInMonths);
		setInterestStrategy(InterestStrategy.COMPOUND);
	}

	@Override
	public double getInterest(int term) {
		return this.getInterestStrategy().getInterest(accountType, amount, term);
	}

}
```

- Préstamo personal.

```java
package org.trishinfotech.abstractfactory.acct.loan;

import org.trishinfotech.abstractfactory.acct.AccountType;
import org.trishinfotech.abstractfactory.acct.InterestStrategy;

public class PersonalLoanAccount extends LoanAccount {

	public PersonalLoanAccount(long accountNo, String accountHolderName, int termInMonths) {
		super(accountNo, accountHolderName, AccountType.PERSONAL_LOAN, termInMonths);
		setInterestStrategy(InterestStrategy.COMPOUND);
	}

	@Override
	public double getInterest(int term) {
		return this.getInterestStrategy().getInterest(accountType, amount, term);
	}

}
```

En lo que respecta a las desventajas de haber usado este patrón se podrían encontrar las siguientes:

- Su implementación puede aumentar la complejidad del código. Se requiere definir múltiples clases abstractas e interfaces, lo que puede resultar en un código más extenso y difícil de comprender.
- Aunque el patrón promueve el desacoplamiento del código cliente de las clases concretas, existe un acoplamiento entre las clases concretas y las fábricas abstractas. Esto significa que, si se agrega una nueva clase de cuenta, se deben realizar cambios en todas las fábricas concretas existentes para admitir la nueva cuenta.

Me parece que una de las posibles alternativas que pudo usar el desarrollador sería utilizar el patrón “Observer”, este resuelve el problema de mantener una relación de uno a muchos entre objetos, de manera que cuando un objeto cambie de estado, todos los objetos dependientes sean notificados y actualizados automáticamente.