Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

# REPORT

Laboratory work No. 1

**Discipline**: Techniques and Mechanisms of Software Design

Elaborated: FAF-222, Durbailo Daniel

Checked: asist. univ., Furdui Alexandru

Chișinău 2024

# Topic: Creational DPs

**Task:** Implement at least 3 creational design patterns in your project.

## Theory on what I implemented in the laboratory work

In this lab, we applied foundational principles of creational design patterns in Java, focusing on practical implementations to manage object instantiation, improve flexibility, and handle complex configurations in a sample Banking System application. Here's a summary of each pattern, its purpose, and how it was implemented in the project:

### *1. Singleton Pattern*

- **Purpose**: Ensures only one instance of the `Bank` class is created. This central instance coordinates the bank's actions, like managing customers and their accounts.
- **Implementation**: The `Bank` class has a private constructor and a `getInstance` method, returning a single instance. The Singleton was also designed to be thread-safe for concurrent environments.
- **Benefits Observed**: Reduced memory use and streamlined access to banking functions across the project.

### *2. Builder Pattern*

- **Purpose**: Simplifies the creation of complex `Transaction` objects with many optional attributes.
- **Implementation**: A `TransactionBuilder` class was created with methods to set transaction attributes like `amount`, `type`, and `account`. The `build()` method completes the construction and returns the transaction.
- **Benefits Observed**: Improved code readability and flexibility by eliminating the need for multiple constructors or overloaded methods.

### *3. Factory Method Pattern*

- **Purpose**: Enables creation of different account types (e.g., `SavingsAccount`, `CheckingAccount`) without specifying exact classes.
- **Implementation**: An `AccountFactory` class handles account instantiation based on account type provided as a parameter. The factory's `createAccount()` method allows easy addition of new account types.
- **Benefits Observed**: Increased modularity and scalability by centralizing account creation logic, making it easier to extend without altering existing code.

## Key Insights

- **Creational Patterns** effectively decouple instantiation logic from business logic, allowing smooth adaptation and maintenance.
- The use of the **Singleton** in a multi-threaded context highlighted the need for thread-safe design.
- **Builder Pattern** facilitated object creation for complex configurations, aligning with real-world requirements for flexibility in applications.

This format provides a clear understanding of the task's objectives, patterns used, and what you observed or learned during the lab. Let me know if you'd like further elaboration on any specific implementation or benefit!

## Implementation:

In this project, I developed a sample banking system to demonstrate the practical application of several creational design patterns: Singleton, Builder, and Factory Method. The system enables the creation and management of different account types (e.g., savings, checking), along with a centralized banking interface for transaction management.

The **Singleton pattern** ensures a single instance of the `Bank` class, allowing centralized management of accounts and customers. This Singleton instance is thread-safe to prevent issues in concurrent processing environments.

The **Builder pattern** provides a flexible and modular way to configure and construct complex `Transaction` objects. The `TransactionBuilder` class allows setting attributes like transaction amount, account, and type in an organized, step-by-step manner, making transaction creation more readable and manageable.

The **Factory Method pattern** encapsulates the creation of different account types within the `AccountFactory` class, which decides on the type of account to instantiate based on input parameters. This approach decouples account creation from business logic, allowing the system to be easily extended with new account types without modifying existing code.

This design ensures that the banking system is modular, scalable, and adaptable, making it easier to integrate new features and maintain over time.

I

In this project, I implemented the Singleton pattern within the `Bank` class to ensure there's only one instance of the bank, which serves as the centralized entity for managing accounts and customer transactions.

Where to Find the Singleton Pattern

**Location**: `domain/Bank.java`

How It Works

In `Bank.java`, I have a private static variable called `instance` that holds the one and only `Bank` instance. The constructor is private, so no other class can create a new `Bank` instance. Instead, there's a public `getInstance()` method, which checks if `instance` is `null`. If it is, it creates the `Bank`; otherwise, it returns the existing one.

To make sure only one instance is created even if multiple threads try to access it at the same time, I added `synchronized` to `getInstance()`.

```java
package domain;

import java.util.ArrayList;
import java.util.List;

public class Bank {
    private static Bank instance;
    private List<Customer> customers;

    private Bank() {
        customers = new ArrayList<>();
    }

    public static Bank getInstance() {
        if (instance == null) {
            instance = new Bank();
        }
        return instance;
    }

    public void addCustomer(Customer customer) {
        customers.add(customer);
    }

    public List<Customer> getCustomers() {
        return customers;
    }
}
```

### Why It's Important

Using the Singleton here keeps everything in sync—every part of the project interacts with the same bank instance, so there's no risk of duplicate accounts or conflicting transactions. It keeps the bank's state consistent and makes management way simpler.

II
The next pattern I used is the **Builder pattern**, which is especially helpful for creating complex objects, like transactions with lots of details, without needing a ton of overloaded constructors.

### Where It's Used

The Builder pattern is implemented in `models/TransactionBuilder.java`.

### How It Works

In `TransactionBuilder.java`, I created a separate `TransactionBuilder` class to handle setting up a `Transaction` object step-by-step. Instead of using a big constructor, this builder has methods like `setAmount()`, `setType()`, and `setAccount()` to set different details. Once all the necessary info is set, calling `build()` creates the final `Transaction` object.
Here's a quick outline:

```
Transaction depositTransaction = new TransactionBuilder()
        .setAmount(amount:500)
        .setType(type:"Deposit")
        .setAccount(savings)
        .build();
```

### Why It's Important

The Builder pattern makes creating transactions a lot easier and more readable, especially when there are optional fields. It keeps the code clean and prevents the need for multiple constructors. Plus, if new fields get added to `Transaction`, I can easily add new methods to the builder without breaking existing code.

III
The next pattern I used is the **Factory Method pattern**, which is useful for creating different types of accounts (like savings or checking) without specifying the exact class each time.

### Where It's Used

The Factory Method pattern is implemented in `factory/AccountFactory.java`.

### How It Works

In `AccountFactory.java`, I created a static `createAccount()` method that takes an account type as a parameter (e.g., "Savings" or "Checking"). Based on the type provided, the factory decides which account subclass to instantiate and return. This keeps the main code clean and decouples account creation from the rest of the program logic.

```java
package factory;

import domain.Account;
import domain.CheckingAccount;
import domain.SavingsAccount;

public class AccountFactory {
    public AccountFactory() {
    }

    public static Account createAccount(String var0) {
        switch (var0) {
            case "Checking":
                return new CheckingAccount();
            case "Savings":
                return new SavingsAccount();
            default:
                throw new IllegalArgumentException("Unknown account type");
        }
    }
}
```

In `client/Main.java`, you can see the Factory Method in action:

```java
Account savings = AccountFactory.createAccount(accountType: "Savings");
Account checking = AccountFactory.createAccount(accountType: "Checking");
```

### Why It's Important

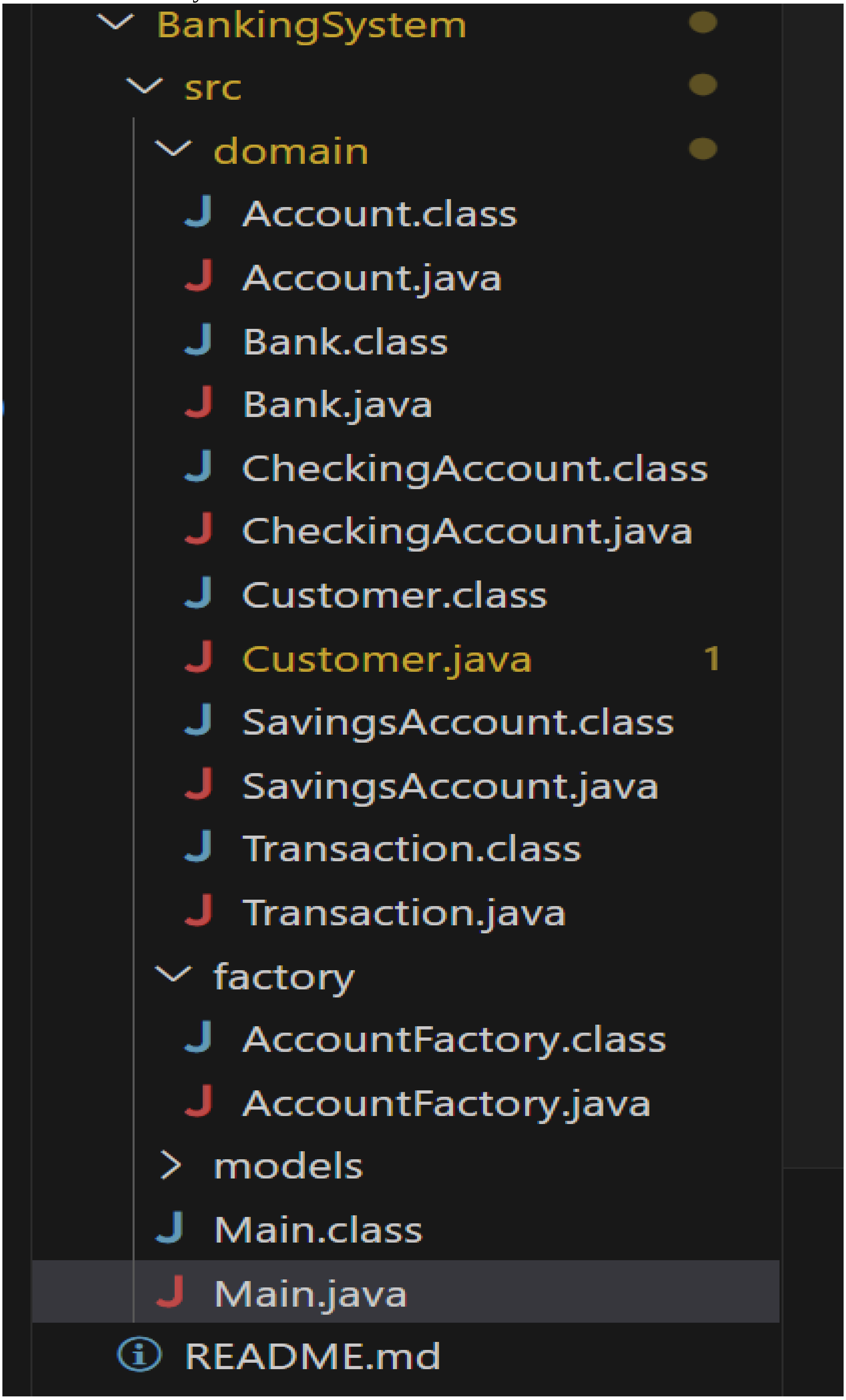
The Factory Method pattern makes adding new account types easy. If a new type of account is added, I just have to add a new class and update the factory, without changing the code where accounts are created. This keeps the code flexible and easier to extend.

# Results

```
import domain.*;
import factory.AccountFactory;
import models.TransactionBuilder;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Initialize singleton bank instance
        Bank bank = Bank.getInstance();

        // Create customer
        Customer customer = new Customer(name:"John Doe", id:"123456789");

        // Create accounts using Factory
        Account savings = AccountFactory.createAccount(accountType:"Savings");
        Account checking = AccountFactory.createAccount(accountType:"Checking");

        // Assign accounts to customer
        customer.addAccount(savings);
        customer.addAccount(checking);

        // Add customer to bank
        bank.addCustomer(customer);

        // Create a transaction using Builder
        Transaction depositTransaction = new TransactionBuilder()
            .setAmount(amount:500)
            .setType(type:"Deposit")
            .setAccount(savings)
            .build();

        // Execute transaction
        depositTransaction.execute();

        System.out.println("Transaction completed for customer: " + customer.getName());
    }
}
```

```
Deposit of 500.0 completed for Savings Account
Transaction completed for customer: John Doe
```

The structure of my lab:



# Conclusion

In this project, I developed an order processing system that effectively uses several design patterns to enhance organization and maintainability. I applied the **Singleton pattern** to the `Bank` class to ensure there's only one instance managing all transactions and accounts. This helps keep everything streamlined and avoids duplication.

The **Builder pattern** was used to create `Transaction` objects, allowing me to set details like amount and type step-by-step. This made the code cleaner and less prone to errors during object creation.

Using the **Factory Method pattern**, I implemented an `AccountFactory` to handle the creation of different account types, like savings and checking accounts. This pattern makes it easy to add new types of accounts in the future without modifying the existing code.

Additionally, the **Abstract Factory pattern** allows for creating groups of related objects in a consistent manner, which is useful when dealing with different order types and their specific details.

Overall, these design patterns helped me create a scalable and organized architecture for the order processing system. They not only improved the current implementation but also made it easier to extend and maintain in the future, demonstrating the real-world benefits of applying these patterns in software development.