Eric Gurevich

Applying Software Design Patterns in the Classroom

I. Abstract

This document explains how to apply software design patterns in an educational context with examples of Java code.

II. Introduction

Design patterns are a crucial, yet often underemphasized concept within the higher-level considerations of designing and implementing effective software. In higher education, courses designed for students in information technology and computer science often thoroughly discuss low-level fundamentals like algorithms, data structures, discrete mathematics, etc., but may fail to explain high-level techniques and principles in the macrocosm of programming modern software. Within the software industry, engineers regularly employ design patterns for several reasons (1).

Firstly, design patterns promote reusable, flexible, modular code, speeding up the development process, compared to blindly programming on the fly and refactoring a system as it gets more complex. Secondly, they provide an established and tested set of ideas and vocabularies that allow for precise, effective communication between involved parties. (4) Thirdly, and most critically, they are descriptive rather than prescriptive. Specifically, they are ascribed from existing solutions to common problems that have already occurred and are likely to reoccur. Thus, design patterns are learned not simply to know them, but to apply them where necessary (5).

This paper covers a few of these patterns that the author believes most useful and relevant to a senior-level software engineering student as applied to programming projects in his or her academic career.

III. Background

Design patterns can be defined as a systemic name, motivation, and explanation of a solution to a recurring design problem in object-oriented systems. A design pattern describes the problem, the solution, when to apply it, and its consequences. These patterns have consistent formats: names, intents, motivations, applications, structure, participants, collaborations, consequences, implementations, samples, uses, and related patterns (5). However, they will be simplified and put into practical use for the purpose of this paper. Patterns can be classified into three purposes, creational, concerning the creation of an object, structural, concerning the way objects inherit from, depend on, and relate to each other, and behavioral, concerning the way objects are programmed to behave given certain conditions. By applying these to an object-oriented system, solutions to common problems can be implemented proactively.

Here are some important design patterns that will be covered in this paper.

a. Structural -
    a. Facade
    b. Adapter
    c. Decorator
b. Creational -

        a. Singleton
    c. Behavioral -
        a. Strategy
        b. Observer

Prior work shows that students can be aided in learning design patterns through approaches such as gamification of a learning activity (2) or an artificial-intelligence based assistant that detects when a specific pattern can be applied (3). The methods tested here will require implementing specific requirements for a simple application that will be come up in many software systems. These specifications will suggest the use of certain patterns for efficient and effective code.

IV. Methods

This section assumes familiarity with UML. Follow along with the author code as you consider these solutions (6).

Patterns as applied to central system – Bank

We will be implementing a simple bank system. The initial architecture will just have a Bank and a BankAccount. We are able to create and retrieve bank accounts with an initial balance. Each account has a unique ID. We are able to get the total account value of the bank. We do not need to worry right now about other common bank functions like withdrawing and depositing to an account yet.

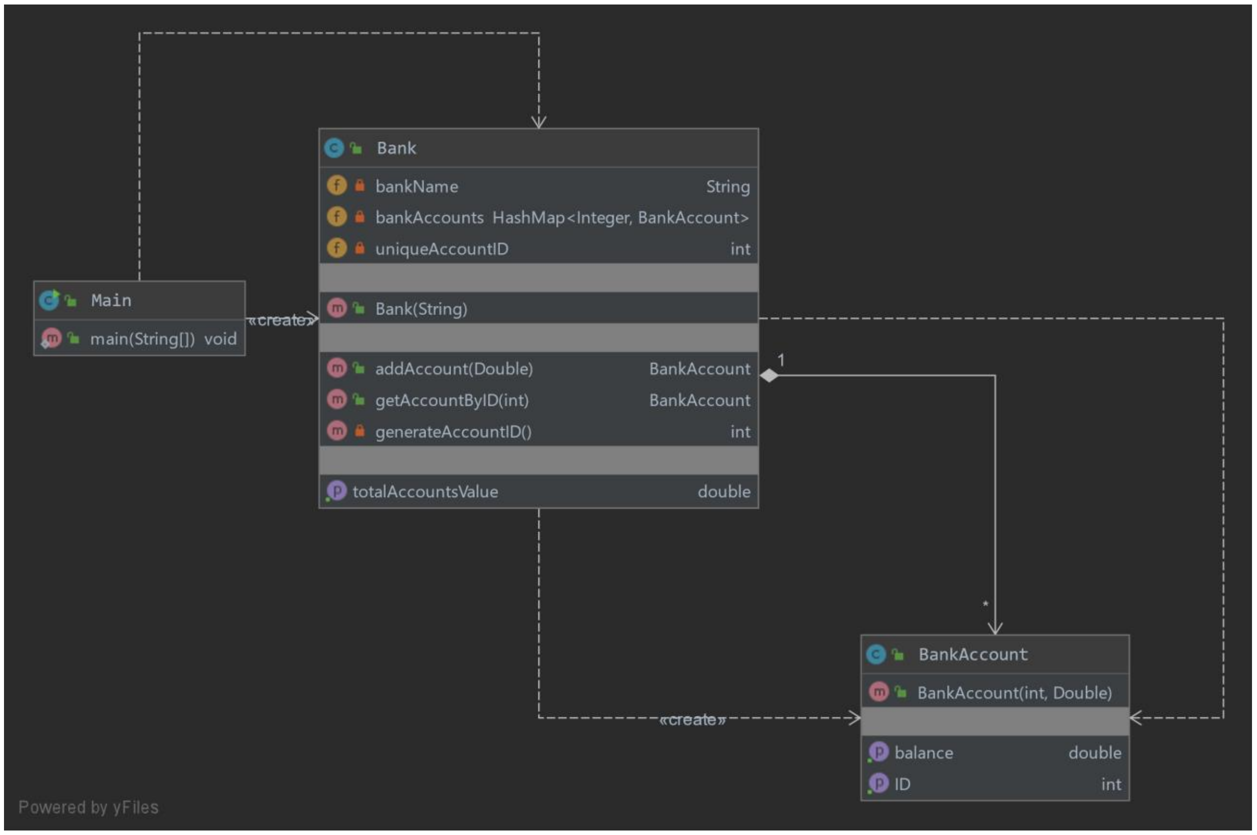


Figure 1 - Initial Bank Architecture

1. **Singleton-** Accessing central bank from several users.

   Our company has specified new requirements. A User must be able to register one or more accounts with the bank and print a list of all their accounts with balance. How does a user access the bank? One idea would be a constructor with the Bank as an argument, but what happens if a user wants to change banks? Or perhaps the object instantiating the user does not have access to a Bank object. A setter would introduce the same issues, where another object will be responsible for setting the User’s bank. Imagine having a million users and having to set the same Bank for each one. Every future class accessing the bank will face the same questions. How do we guarantee universal access to our sole bank without dangerous global variables?

   The singleton pattern provides a universal access point to an object to all classes in the package, guaranteeing that only one of the object exists. This is perfect for our single Bank that many classes will need access to. This way each class is only responsible for accessing the bank when it needs to, by simply calling a static method to return the single instance.

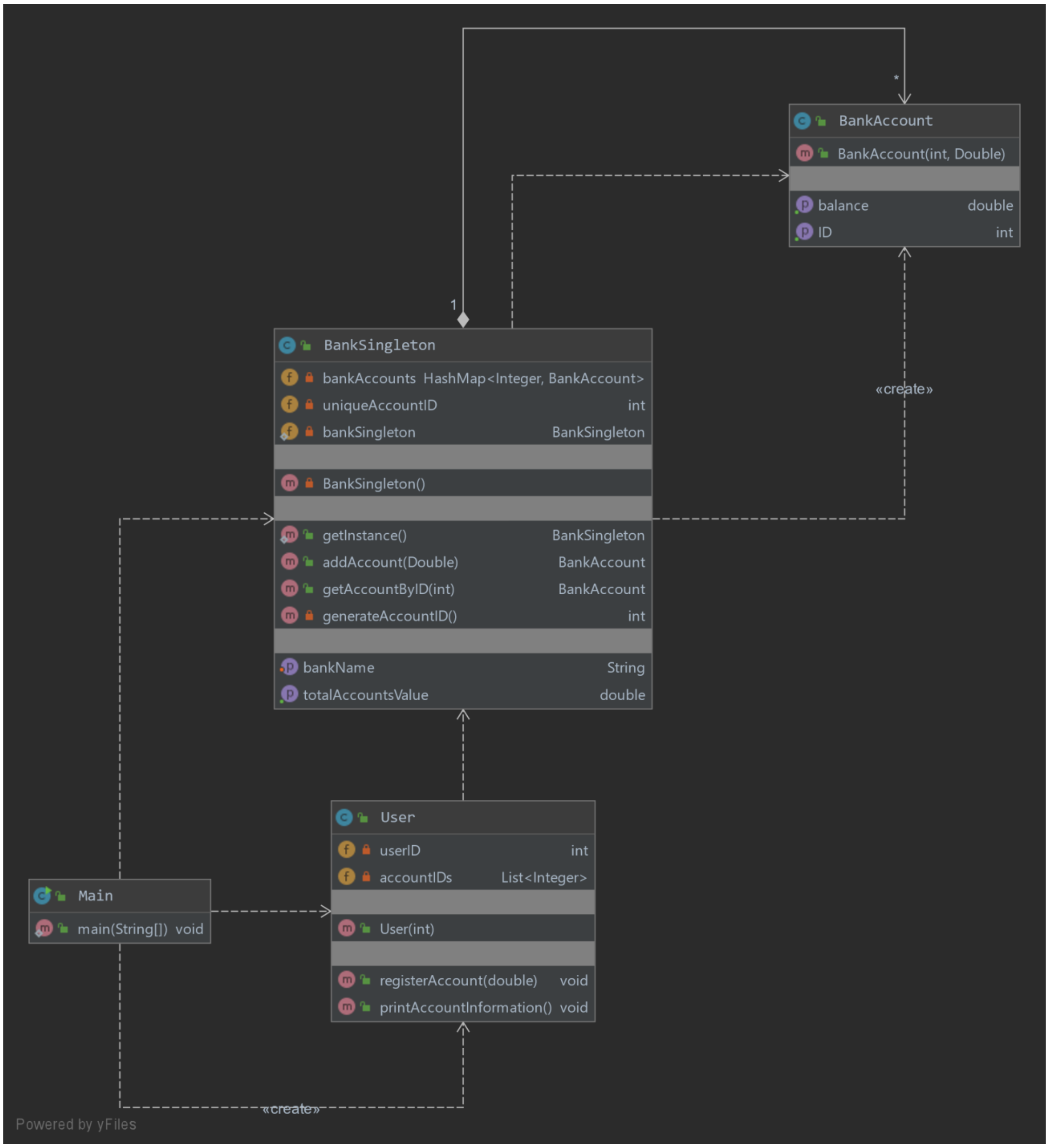


Figure 2 - Singleton Pattern

<u>Implementation:</u> Our singleton class has a private constructor that should only initialize private variables to a default value. The only time this constructor is called is in the static getInstance() which returns either the already initialized instance or initializes a new one if null. Any public methods will be called on the instance and can access its private variables.

```java
public class BankSingleton {
    private String bankName;
    private HashMap<Integer, BankAccount> bankAccounts;
    private int uniqueAccountID;

    private static BankSingleton bankSingleton;

    private BankSingleton(){
        uniqueAccountID = 0;
        bankAccounts = new HashMap<>();
    }

    public static BankSingleton getInstance(){
        if (bankSingleton == null) {
            bankSingleton = new BankSingleton();
        }
        return bankSingleton;
    }

    public void setBankName(String bankName) {
        this.bankName = bankName;
    }

    public BankAccount addAccount(Double initialBalance) {
        int newID = generateAccountID();
        BankAccount newAccount = new BankAccount(newID, initialBalance);
        bankAccounts.put(newID, newAccount);

        return newAccount;
    }

    public BankAccount getAccountByID(int id) {
        return bankAccounts.get(id);
    }

    private int generateAccountID() {
        return ++uniqueAccountID;
    }

    public double getTotalAccountsValue() {
        double sum = 0.00;

        for (BankAccount bankAccount: bankAccounts.values()) {
            sum += bankAccount.getBalance();
        }

        return sum;
    }
}
```

```
public class User {
    private int userID;
    private List<Integer> accountIDs;

    public User(int userID) {
        this.userID = userID;
        accountIDs = new ArrayList<>();
    }

    public void registerAccount(double initialBalance) {

accountIDs.add(BankSingleton.getInstance().addAccount(initialBalance).getID()
);
    }

    public void printAccountInformation() {
        BankSingleton bank = BankSingleton.getInstance();
        for (int accountID: accountIDs) {
            System.out.println("Account " + accountID + " contains $" +
bank.getAccountByID(accountID).getBalance());
        }
    }
}
```

2. **Strategy** - User wants to withdraw money from their account. Depending on size of account, different algorithms for how much can withdraw.

   Our company now wants users to be able to withdraw money, but with a catch. The max amount of money the user can withdraw depends on the account balance. There are three different rules depending on the balance amount. A simple series of if-else conditions may work now, but what about if there ae additional rules implemented later, or if one of the rules has to be modified? Or perhaps another type of account is designed and wants to borrow some of the same rules. This would lead to a refactoring nightmare if all the rules are contained in one class.

   We can use the strategy pattern here to make our lives easier. An interface with a calculateMaxWithdraw method will be implemented by three different subclasses that can be instantiated as needed by the caller, in our case, the Account’s withdraw method. This way, the Account must call only one method no matter the rule, and rules can be swapped out, added, or removed easily.

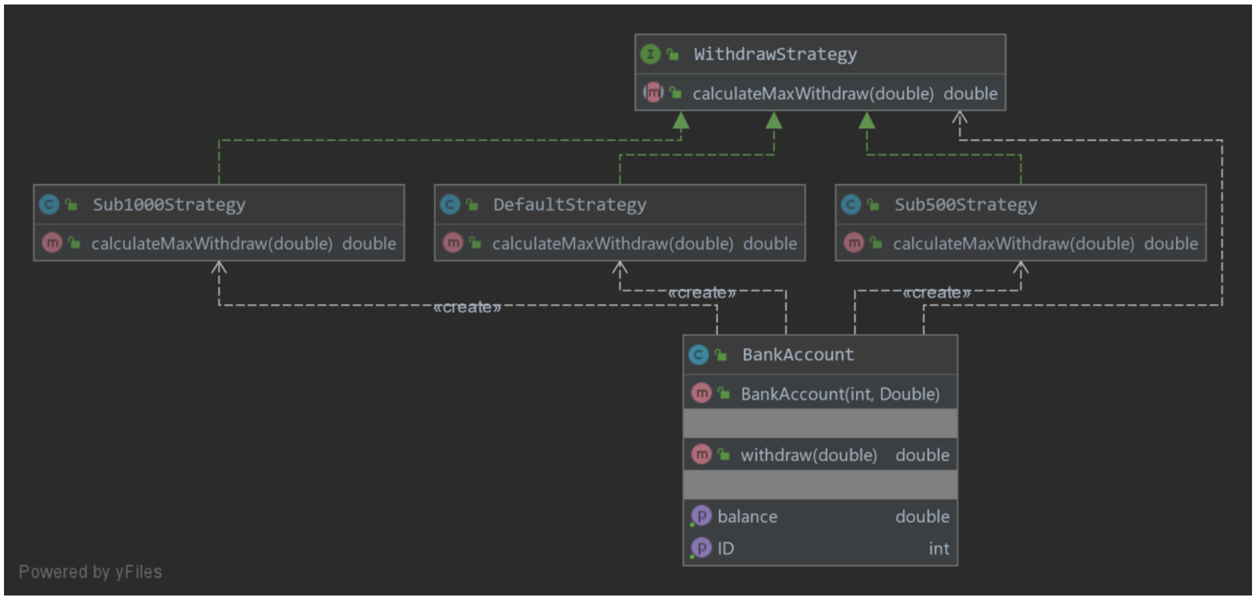


Figure 3 - Strategy Pattern

Implementation: This one is very simple but powerful. We need an interface with the methods we might use between each strategy, then implement those methods for each strategy. Our strategy is implemented by three classes, Sub500Strategy, Sub1000Strategy, and DefaultStrategy. Then in the calling class, a WithdrawStrategy is created and assigned to one of three implementation classes depending on the account balance. Then all we need to do is call the calculateMaxWithdraw() method to obtain a return value we can use further.

```
public interface WithdrawStrategy {
    double calculateMaxWithdraw(double balance);
}
```

```
public double withdraw(double amount) {
        if (amount > balance) {
            throw new IllegalArgumentException("Cannot withdraw more than 
balance.");
        }

        WithdrawStrategy withdrawStrategy;

        if (balance < 500) {
            withdrawStrategy = new Sub500Strategy();
        } else if (balance < 1000) {
            withdrawStrategy = new Sub1000Strategy();
        } else {
            withdrawStrategy = new DefaultStrategy();
        }

        double maxWithdraw = withdrawStrategy.calculateMaxWithdraw(balance);

        if (amount > maxWithdraw) {
            throw new IllegalArgumentException("Cannot withdraw more than $" 
+ maxWithdraw);
        }

        balance -= amount;
        return amount;
    }
```

3. **Decorator** – Extend the features and functionality of a bank account without a complicated web of inheritance.

   Our superiors want to support multiple types of accounts with multiple features. For example, an account could be singular or joint, and has the option of being checking, savings, or high interest.

   We could do this by creating a common abstract superclass or interface, but then each time we change the superclass, we have to update the implementations. Furthermore, to combine features, we need a separate class for each combination feature (e.g. we would need JointChecking, JoinSavings, and JoinHighInterest classes). This leads to a nightmare of inheritance. A better way is to have a separate decorator for each feature that contains an instance of the decorated object, without having to worry what already is or is not decorated.

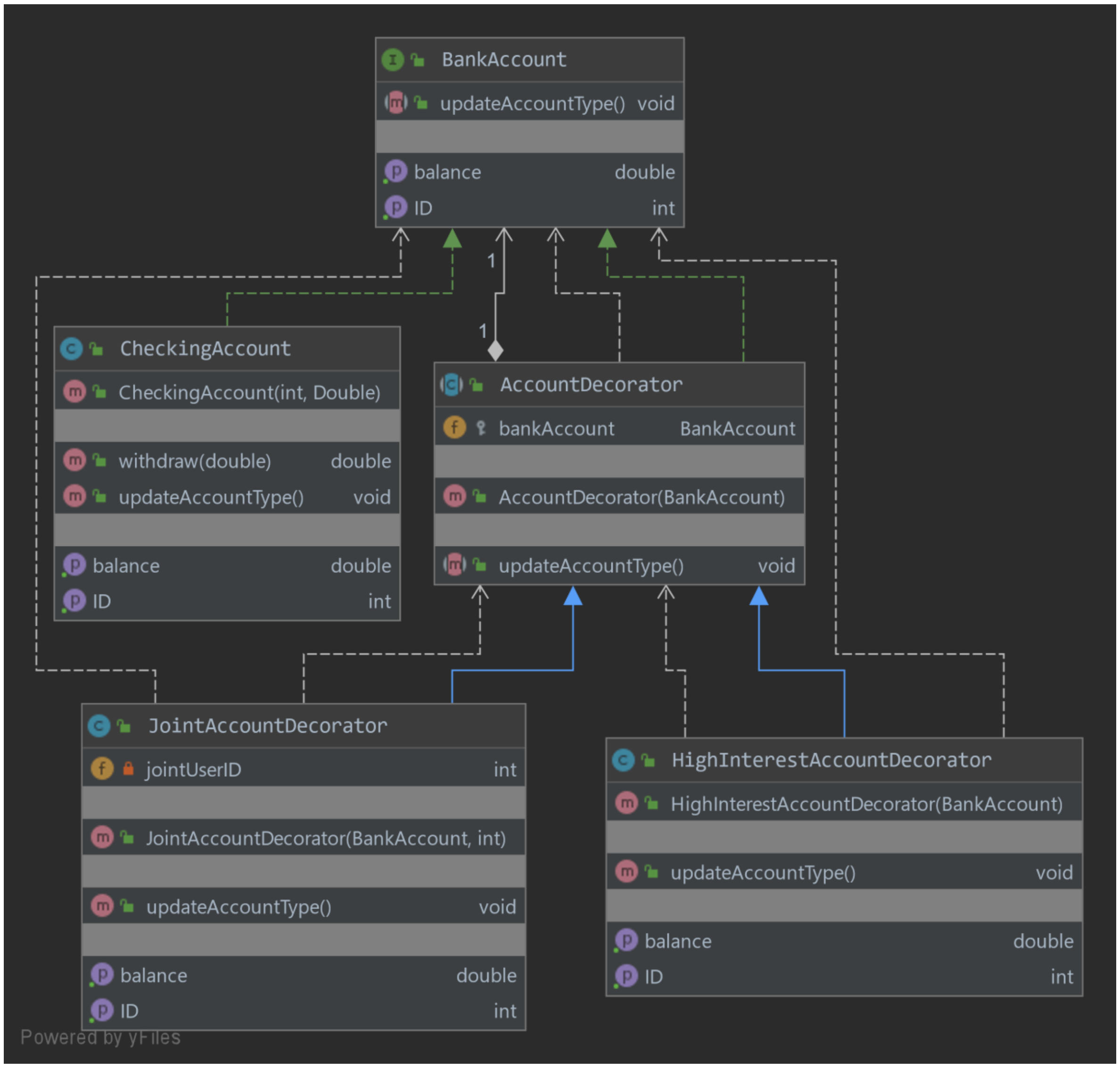


Figure 4 - Decorator Pattern

<u>Implementation:</u> In this pattern, rather than rely on inheritance to add features to an object, a wrapper class is used to decorate each object with a specific feature. AccountDecorator inherits from BankAccount which is generified into an interface with the methods we need, and updateAccountType() is abstract. Notice that AccountDecorator contains an instance of BankAccount, and a constructor in which it is passed in. This is the instance that we are adding on to, and can be decorated by each decorator as many times as we want. AccountDecorator is subclassed into two decorators that concretely implement updateAccountType(). These implementations can do whatever you want to the BankAccount instance. In fact, you can even add another field and constructor argument to add additional functionality that the client code can call when decorating.

The client code simply instantiates a concrete BankAccount, in this case a CheckingAccount, then passes it in through each decorator’s constructor along with any additional arguments. We end up with the original CheckingAccount decorated by cohesive and flexible classes that only have one job each!

```
public interface BankAccount {
    double getBalance();
    int getID();
    void updateAccountType();
}

public class CheckingAccount implements BankAccount{

    @Override
    public void updateAccountType() {
        System.out.println("This is a checking account.");
    }
}
public abstract class AccountDecorator implements BankAccount {
    protected BankAccount bankAccount;

    public AccountDecorator(BankAccount bankAccount) {
        this.bankAccount = bankAccount;
    }

    public abstract void updateAccountType();
}

public class HighInterestAccountDecorator extends AccountDecorator {
    public HighInterestAccountDecorator(BankAccount bankAccount) {
        super(bankAccount);
    }

    @Override
    public double getBalance() {
        return bankAccount.getBalance();
    }

    @Override
    public int getID() {
        return bankAccount.getID();
    }

    @Override
    public void updateAccountType() {
        bankAccount.updateAccountType();
        System.out.println("This is a high interest account.");
    }
}
```

```
public class JointAccountDecorator extends AccountDecorator{
    private int jointUserID;

    public JointAccountDecorator(BankAccount bankAccount, int jointUserID) {
        super(bankAccount);
        this.jointUserID = jointUserID;
    }

    @Override
    public double getBalance() {
        return bankAccount.getBalance();
    }

    @Override
    public int getID() {
        return bankAccount.getID();
    }

    @Override
    public void updateAccountType() {
        bankAccount.updateAccountType();
        System.out.println("This is a joint account with User: " +
jointUserID);
    }
}
public class Main {
    public static void main(String[] args) {
        BankSingleton bank = BankSingleton.getInstance();

        BankAccount bankAccount = bank.addCheckingAccount(1000.00);
        bankAccount = new HighInterestAccountDecorator(bankAccount);
        bankAccount = new JointAccountDecorator(bankAccount, 1);

        bankAccount.updateAccountType();
    }
}
```

4. **Observer -** Notify multiple accounts simultaneously that an interest payment has been received.

   Our bank must pay our users interest on the balance they store in their Account. Rather than go account by account and calculate how much we owe them, why don’t we send a message to every Account with the current interest rate and have them calculate themselves how much the bank has to deposit?

   We can implement a class that is responsible for registering listeners, then when triggered by client code, can update each listener with a message. This takes away responsibility from each listening class, and prevents constant queries from slowing down the central class.

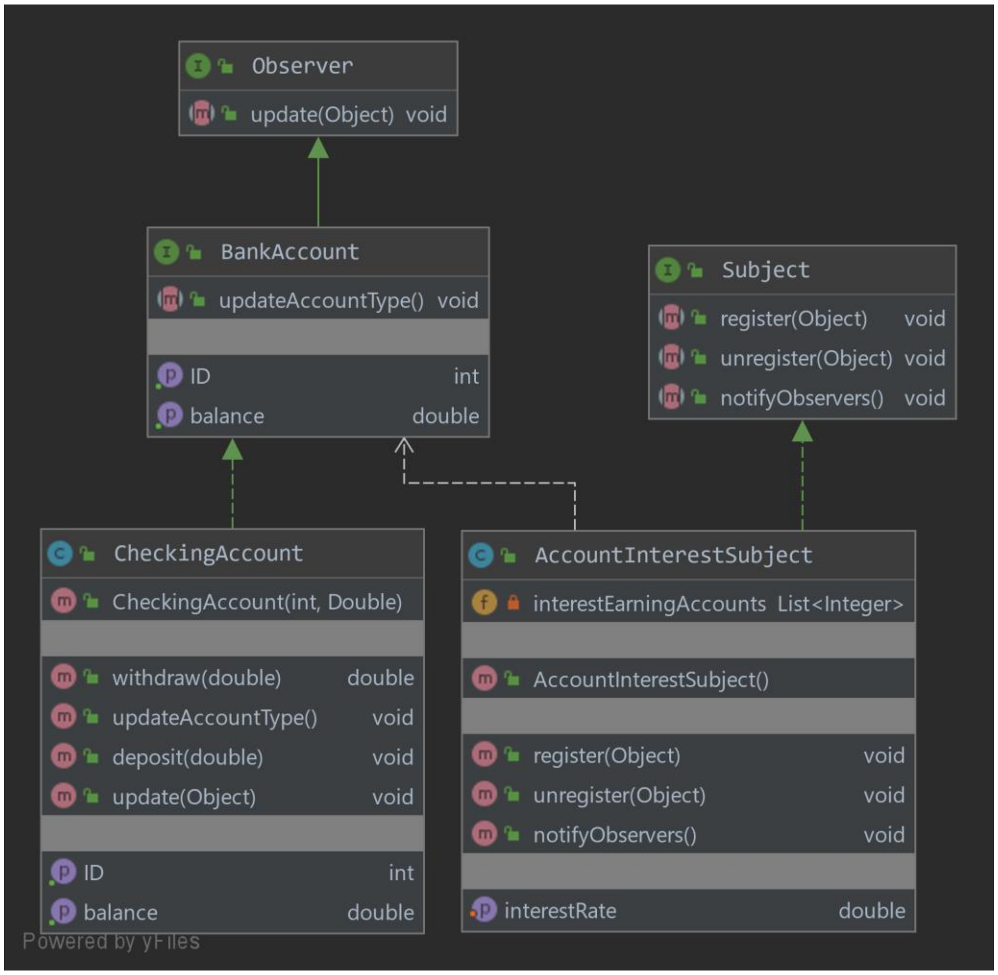


Figure 5 - Observer Pattern

Implementation: To start, create a Subject interface with register, unregistered, and notify methods. Create an Observer interface with the method triggered from the Subject. In the concrete implementation of the Subject, register and unregister will add or remove to/from some collection of Observers. The notify method will iterate through each and send a message through the update() method. In the concrete Observer implementation, the update() method will pass in some object. When notifyObservers() is called, our AccountInterestSubject will send the current interest rate to each CheckingAccount, which will use that to update the balance with the correct interest.

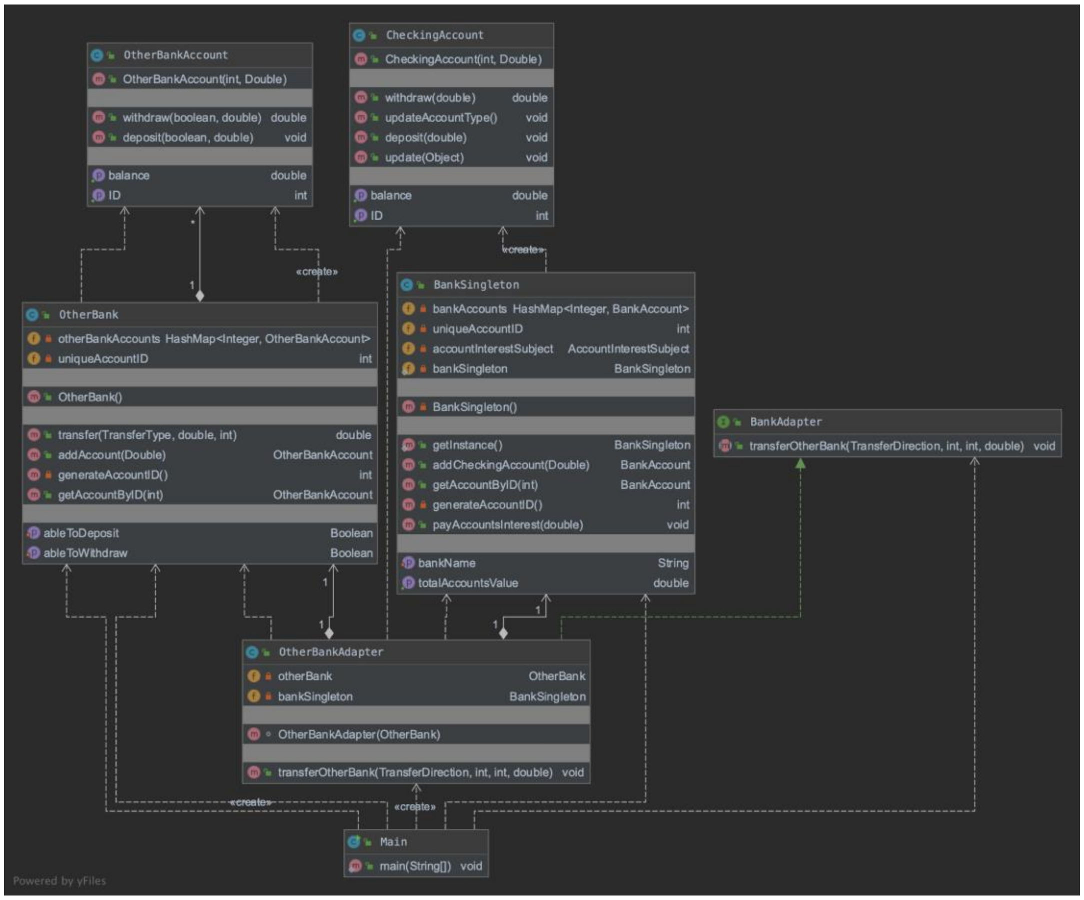
```
public interface Observer {
    void update(Object message);
}

public interface Subject {
    void register(Object observer);
    void unregister(Object observer);

    void notifyObservers();
}
```

```
public interface BankAccount extends Observer{
    double getBalance();
    int getID();
    void updateAccountType();
}

public class AccountInterestSubject implements Subject {
    private List<Integer> interestEarningAccounts;
    private double interestRate;

    public AccountInterestSubject() {
        interestRate = 0;
        interestEarningAccounts = new ArrayList<>();
    }

    @Override
    public void register(Object accountID) {
        interestEarningAccounts.add((Integer)accountID);
    }

    public void unregister(Object accountID) {
        interestEarningAccounts.remove(accountID);
    }

    public void notifyObservers() {
        BankAccount bankAccount;
        for (int id : interestEarningAccounts) {

BankSingleton.getInstance().getAccountByID(id).update(interestRate);
        }
    }

    public void setInterestRate(double interestRate) {
        this.interestRate = interestRate;
    }
}
public class CheckingAccount implements BankAccount {
    public void deposit(double amount) {
        balance += amount;
    }

    @Override
    public void update(Object interestRate) {
        deposit(getBalance() * (Double) interestRate);
    }
}
```

```
public class BankSingleton {
    private AccountInterestSubject accountInterestSubject;

    public BankAccount addCheckingAccount(Double initialBalance) {
        int newID = generateAccountID();
        BankAccount newAccount = new CheckingAccount(newID, initialBalance);
        bankAccounts.put(newID, newAccount);

        accountInterestSubject.register(newID);

        return newAccount;
    }

    public void payAccountsInterest(double interestRate) {
        accountInterestSubject.setInterestRate(interestRate);
        accountInterestSubject.notifyObservers();
    }
}


public class Main {
    public static void main(String[] args) {
        BankSingleton bank = BankSingleton.getInstance();

        BankAccount bankAccount1 = bank.addCheckingAccount(1000.00);
        BankAccount bankAccount2 = bank.addCheckingAccount(2000.00);

        bank.payAccountsInterest(0.05);

        System.out.println(bankAccount1.getBalance());
        System.out.println(bankAccount2.getBalance());
    }
}
```

5. **Adapter** – Implement a middleman class to simplify transferring money with another very different bank account.

   Even though our users rely on our bank, they may want to transfer funds between accounts belonging to different banks. In order to do this manually, they would have to navigate different, complex interfaces that adhere to each bank. Or, we could create an adapter class that will act as the middleman allowing one or two-way conformation between the banks' different transfer methods. This way the client code is not responsible for implementing an adapter per each transfer, preventing violation of the single-responsibility principle.


OtherBankAccount
OtherBankAccount(int, Double)
withdraw(boolean, double) double
deposit(boolean, double) void
balance double
ID int
CheckingAccount
CheckingAccount(int, Double)
withdraw(double) double
updateAccountType() void
deposit(double) void
update(Object) void
balance double
ID int
«create»
OtherBank
otherBankAccounts HashMap<Integer, OtherBankAccount>
uniqueAccountID int
OtherBank()
transfer(TransferType, double, int) double
addAccount(Double) OtherBankAccount
generateAccountID() int
getAccountByID(int) OtherBankAccount
ableToDeposit Boolean
ableToWithdraw Boolean
BankSingleton
bankAccounts HashMap<Integer, BankAccount>
uniqueAccountID int
accountInterestSubject AccountInterestSubject
bankSingleton BankSingleton
BankSingleton()
getInstance() BankSingleton
addCheckingAccount(Double) BankAccount
getAccountByID(int) BankAccount
generateAccountID() int
payAccountsInterest(double) void
bankName String
totalAccountsValue double
BankAdapter
transferOtherBank(TransferDirection, int, int, double) void
OtherBankAdapter
otherBank OtherBank
bankSingleton BankSingleton
OtherBankAdapter(OtherBank)
transferOtherBank(TransferDirection, int, int, double) void
Main
main(String[]) void
Powered by yFiles

<u>Implementation:</u> In our OtherBank class, the transfer() method transfers funds into or out of an OtherBankAccount. This is different from our SingletonBank system in which the User class is responsible for withdraws and deposits by the client. In addition the transfer() method requires an enum value for transfer type—deposit or withdraw. This method calls the account's withdraw and deposit methods, passing in a Boolean flag indicating whether the type of transfer is allowed. As you can see there are various irregularities between the two bank architectures. We don't want the client code responsible for translation, nor our Bank or User class, as they have their own jobs to do.

We will implement an adapter class. The concept here is simply to write methods that allow the two classes to work with each other in a logical way, while the adapter takes care of the details. Our BankAdapter interface has a method transferOtherBank() implemented by OtherBankAdapter, which takes in an OtherBank through its constructor and accesses our bank through SingletonBank. The method requires a transfer direction, either to our bank or to OtherBank, allowing for two-way translation. Besides this, the IDs of both bank accounts, and transfer amount are passed in. Our client code can call this adapter class method and ignore any gory details regarding the implementation.

```java
public class OtherBank {
    private HashMap<Integer, OtherBankAccount> otherBankAccounts;
    private int uniqueAccountID;
    private boolean ableToWithdraw;
    private boolean ableToDeposit;

    public enum TransferType {
        WITHDRAW,
        DEPOSIT
    }

    public OtherBank() {
        uniqueAccountID = 0;
        otherBankAccounts = new HashMap<>();
        ableToWithdraw = false;
    }

    public double transfer(TransferType transferType, double amount, int accountID) {
        switch (transferType) {
            case WITHDRAW:
                if (!otherBankAccounts.containsKey(accountID)) {
                    throw new IllegalArgumentException("Account does not exist.");
                }

                return getAccountByID(accountID).withdraw(ableToWithdraw, amount);
            case DEPOSIT:
                if (!otherBankAccounts.containsKey(accountID)) {
                    throw new IllegalArgumentException("Account does not exist.");
                }

                getAccountByID(accountID).deposit(ableToDeposit, amount);
                return 0;
            default:
                throw new IllegalArgumentException("No transfer type.");
        }
    }

    public void setAbleToWithdraw(Boolean ableToWithdraw) {
        this.ableToWithdraw = ableToWithdraw;
    }
}
```

```java
public class OtherBankAccount {
    private int id;
    private Double balance;

    public OtherBankAccount(int id, Double initialBalance) {
        this.id = id;
        balance = initialBalance;
    }

    public double withdraw(boolean ableToWithdraw, double amount) {
        if (!ableToWithdraw) {
            throw new IllegalArgumentException(”Not able to withdraw.”);
        }

        if (amount > balance) {
            throw new IllegalArgumentException(”Cannot withdraw more than balance.”);
        }

        balance -= amount;
        return amount;
    }

    public void deposit(boolean ableToDeposit, double amount) {
        if (!ableToDeposit) {
            throw new IllegalArgumentException(”Not able to deposit.”);
        }

        balance += amount;
    }

    public int getID() {
        return id;
    }

    public double getBalance() {
        return balance;
    }
}
```

```
public class OtherBankAdapter implements BankAdapter{

    private OtherBank otherBank;

    private BankSingleton bankSingleton;


    OtherBankAdapter(OtherBank otherBank) {

        this.otherBank = otherBank;

        bankSingleton = BankSingleton.getInstance();

    }


    public void transferOtherBank(TransferDirection transferDirection, int
bankAccountID, int otherBankAccountID,

                                  double amount) {

        switch (transferDirection) {

            case TO_OTHER_BANK:

                otherBank.setAbleToDeposit(true);

                ((CheckingAccount)
bankSingleton.getAccountByID(bankAccountID)).withdraw(amount);

                otherBank.transfer(OtherBank.TransferType.DEPOSIT, amount,
otherBankAccountID);

                break;

            case FROM_OTHER_BANK:

                otherBank.setAbleToWithdraw(true);

                otherBank.transfer(OtherBank.TransferType.WITHDRAW, amount,
otherBankAccountID);

                ((CheckingAccount)
bankSingleton.getAccountByID(bankAccountID)).deposit(amount);

                break;

        }

    }

}
```

```
public class Main {

    public static void main(String[] args) {

        BankSingleton bank = BankSingleton.getInstance();

        int bankAccountID = bank.addCheckingAccount(1000.00).getID();


        OtherBank otherBank = new OtherBank();

        int otherBankAccountID = otherBank.addAccount(2000.00).getID();


        BankAdapter bankAdapter = new OtherBankAdapter(otherBank);

        bankAdapter.transferOtherBank(BankAdapter.TransferDirection.TO_OTHER_BANK,
bankAccountID, otherBankAccountID, 500.00);


        System.out.println(bank.getAccountByID(bankAccountID).getBalance());

System.out.println(otherBank.getAccountByID(otherBankAccountID).getBalance());

    }

}
```

6. **Façade** - Encapsulate everything into a simple-to-use interface.

   We now have many different classes, objects, and methods within our codebase. In order to use our bank, we must instantiate many objects and set them up them with any dependencies and fields they may require. For the average user of a bank this is a complicated task they probably don't want to worry about every time they go to use their accounts.

   We can now use a design pattern that you have probably employed before without realizing. The façade pattern is a central class that acts as an interface between multiple classes and objects and the client. The façade should contain easy to use methods that take care of the fine details between constructing and using dependencies to achieve something the user will do often. For example, our bank façade should be able to do the following:
   - Connect to our bank
   - Register user and return user ID
   - Get user's account ids
   - Withdraw checking
   - Deposit checking
   - Decorate account
   - Update interest for accounts
   - Transfer checking to another bank

   As a reader exercise, see if you can implement these methods in a façade class.

Sources

1. Kent Beck, Ron Crocker, Gerard Meszaros, John Vlissides, James O. Coplien, Lutz Dominick, and Frances Paulisch. 1996. Industrial experience with design patterns. In Proceedings of the 18th international conference on Software engineering (ICSE '96). IEEE Computer Society, Washington, DC, USA, 103-114.
2. A. Bartel and G. Hagel, "Gamifying the learning of design patterns in software engineering education," 2016 IEEE Global Engineering Education Conference (EDUCON), Abu Dhabi, 2016, pp. 74-79.
3. Berdun, L., Amandi, A. and Campo, M. (2014), An intelligent tutor for teaching software design patterns. Comput Appl Eng Educ, 22: 583-592.
4. Maioriello, James. “What Are Design Patterns and Do I Need Them?” What Are Design Patterns and Do I Need Them? - Developer.com, Developer.com, 2 Oct. 2002.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
6. Gurevich, Eric. “Designpatterns.” GitHub, https://github.com/ericgurevich/designpatterns.