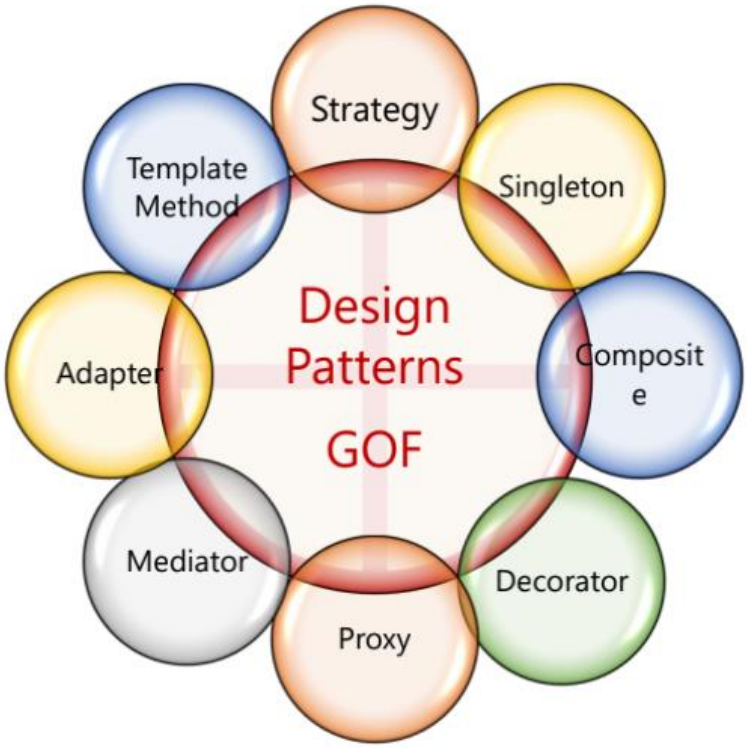
جـامعـة الحـسن الثـاني بالدار البيـضـاء
UNIVERSITÉ HASSAN II DE CASABLANCA

ENSET
ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

Builder, Singleton, Prototype Pattern



Réalisé par : REGOUG Zakia

GLSID3

Année universitaire : 2023-2024

Catégorie :

Création

Exercice :

Développer une application OO qui permet de gérer des comptes bancaires. Chaque Compte est défini par :

- accountId de type String
- balance de type double
- currency : String
- accountType : AccountType (SAVING_ACCOUNT, CURRENT_ACCOUNT)
- accountStatus : AccountStatus (CREATED, ACTIVATED, BLOCKED, SUSPENDED)

Travail demandé :

1. Crée la classe BankAccount

```
public class BankAccount {

    private Long accountId;
    private double balance;
    private String currency;
    private AccountType type;
    private AccountStatus status;

    public BankAccount(Long accountId, double balance, String currency,
AccountType type, AccountStatus status) {
        this.accountId = accountId;
        this.balance = balance;
        this.currency = currency;
        this.type = type;
        this.status = status;
    }

    public BankAccount() {
    }
...
}
```

2. Créer l'interface AccountRepository déclarant les opérations qui permettent de :

- Ajoute run compte
- Consulter tous les comptes
- Consulter un compte sachant son id
- Chercher un compte avec un prédicat quelconque

```
public interface AccountRepository {
    BankAccount save(BankAccount bankAccount);
    List<BankAccount> findAll();
    Optional<BankAccount> findById(Long id); // retourner un compte ou rien
    List<BankAccount> searchAccounts(Predicate<BankAccount>
predicate);//predicate retourne oui ou non
    BankAccount update(BankAccount bankAccount);
    void deleteById(Long id);
}
```

3. Créer une implémentation (BankRepositoryImpl)de cette interface en stockant les comptes dans une collection de type HashMap.

```
public class AccountRepositoryImpl implements AccountRepository {
    private Map<Long, BankAccount> bankAccountMap=new HashMap<>();
    private long accountsCount=0;
    @Override
    public BankAccount save(BankAccount bankAccount) {
        Long accountId =++accountsCount;
        bankAccount.setAccountId(accountId);
        bankAccountMap.put(accountId,bankAccount);
        return bankAccount;
    }

    @Override
    public List<BankAccount> findAll() {
        return bankAccountMap.values().stream().toList();
    }

    @Override
    public Optional<BankAccount> findById(Long id) {
        BankAccount account=bankAccountMap.get(id);
        if(account==null)
            return Optional.empty();
        else
            return Optional.of(account);
    }

    @Override
    public List<BankAccount> searchAccounts(Predicate<BankAccount>
predicate) {
        return
bankAccountMap.values().stream().filter(predicate).collect(Collectors.toLis
t());
    }

    @Override
    public BankAccount update(BankAccount bankAccount) {
        bankAccountMap.put(bankAccount.getAccountId(),bankAccount);
        return bankAccount;
    }

    @Override
    public void deleteById(Long id) {
        bankAccountMap.remove(id);
    }

    public void populateDate(){
        for(int i=0;i<10;i++){

        }
    }
}
```

4. Implémenter le pattern Builder pour la classe Compte

Dans la classe Compte on ajoute une classe statique accountBuilder dont laquel on construit l'objet :

```
public static class AccountBuilder{

    private BankAccount bankAccount=new BankAccount();

    public AccountBuilder accountId(Long id){
```

```
        bankAccount.accountId=id;
        return this;
    }
    public AccountBuilder balance(double balance){
        bankAccount.balance=balance;
        return this;
    }
    public AccountBuilder currency(String currency){
        bankAccount.currency=currency;
        return this;
    }
    public AccountBuilder type(AccountType type){
        bankAccount.type=type;
        return this;
    }
    public AccountBuilder status(AccountStatus status){
        bankAccount.status=status;
        return this;
    }

    public BankAccount build(){
        return bankAccount;
    }
}
```

L'astuce est de retourner this pour permettre de faire .id.balance…

Mais on doit ajouter une méthode builder dans Compte:

```
public static AccountBuilder builder(){
    return new AccountBuilder();
}
```

Exécution :

```
public class Main {
    public static void main(String[] args) {
        BankAccount bankAccount=BankAccount.builder()
                .accountId(1L)
                .balance(4000)
                .currency("MAD")
                .status(AccountStatus.CREATED)
                .type(AccountType.CURRENT_ACCOUNT)
                .build();
        System.out.println(bankAccount.toString());
    }
}
```

```
Main
C:\Users\Zakia\.jdks\corretto-17.0.6\bin\java.exe "-javaagent:C:\Program Files\JetBrains\Intell
BankAccount{accountId=1, balance=4000.0, currency='MAD', type=CURRENT_ACCOUNT, status=CREATED}
```

Dans les bonnes pratiques cette méthode ne doit pas être a l'intérieur de la classe, on déclare une autre classe ou on mettre tous les builder :

```
public class BankDirector {
    1 usage
    public static BankAccount.AccountBuilder accountBuilder(){
        return new BankAccount.AccountBuilder();
    }
}
```

```
        BankAccount bankAccount= BankDirector.accountBuilder()
                .accountId(1L)
                .balance(4000)
                .currency("MAD")
                .status(AccountStatus.CREATED)
                .type(AccountType.CURRENT_ACCOUNT)
                .build();
        System.out.println(bankAccount.toString());
    }
}
```

```
C:\Users\Zakia\.jdks\corretto-17.0.6\bin\java.exe "-javaagent:C:\Program Files\JetBrains\Intel
BankAccount{accountId=1, balance=4000.0, currency='MAD', type=CURRENT_ACCOUNT, status=CREATED}
```

On utilisant le repository :

```
public void populateData(){
    for(int i=0;i<10;i++){
        BankAccount bankAccount= BankDirector.accountBuilder()
                .balance(20000+Math.random()*12000)
                .type(Math.random()>0.5?
AccountType.SAVING_ACCOUNT:AccountType.CURRENT_ACCOUNT)
                .status(Math.random()>0.5?
AccountStatus.CREATED:AccountStatus.ACTIVATED)
                .currency(Math.random()>0.3?"USD":"MAD")
                .build();
        save(bankAccount);
    }
}
```

```
AccountRepositoryImpl accountRepository=new AccountRepositoryImpl();
accountRepository.populateData();
List<BankAccount> bankAccounts=accountRepository.findAll();
bankAccounts.forEach(System.out::println);
```

```
BankAccount{accountId=1, balance=31557.522371281244, currency='MAD', type=CURRENT_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=2, balance=21381.99522748491, currency='MAD', type=CURRENT_ACCOUNT, status=CREATED}
BankAccount{accountId=3, balance=28283.88403985249, currency='USD', type=CURRENT_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=4, balance=29562.72871121223, currency='USD', type=CURRENT_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=5, balance=26002.804599059382, currency='MAD', type=SAVING_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=6, balance=23377.783074843413, currency='USD', type=SAVING_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=7, balance=20581.81190477597, currency='MAD', type=SAVING_ACCOUNT, status=CREATED}
BankAccount{accountId=8, balance=25788.740369622632, currency='MAD', type=CURRENT_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=9, balance=23052.948411491307, currency='USD', type=SAVING_ACCOUNT, status=CREATED}
BankAccount{accountId=10, balance=22151.94381100612, currency='USD', type=CURRENT_ACCOUNT, status=CREATED}
```

5. Implémenter le pattern Singleton pour créer une instance unique de BankRepositoryImpl

On utilise ce pattern dans le cas où on veut interdire d'utiliser new et on veut instancier l'objet une seule fois.

1ere méthode :

```
private static final AccountRepositoryImpl accountRepository;
static {
    System.out.println("Singleton Instatiation ");
    accountRepository=new AccountRepositoryImpl();
}
private AccountRepositoryImpl(){

}
public static AccountRepositoryImpl getInstance(){
    return accountRepository;
}
```

```
AccountRepositoryImpl accountRepository=AccountRepositoryImpl.getInstance();
```

```
Main
C:\Users\Zakia\.jdks\corretto-17.
Singleton Instatiation
```

2eme méthode : sans le block static

```
3 usages
private static  AccountRepositoryImpl accountRepository;

2 usages
public static AccountRepositoryImpl getInstance(){
    if(accountRepository==null) {
        System.out.println("Singleton Instatiation");
        accountRepository = new AccountRepositoryImpl();
    }
    return accountRepository;
}
```

Problème avec thread :

Main :

```
for(int i=0;i<10;i++){
    new Thread(()->{
        accountRepository.populateData();
    }).start();
}
System.out.println("taper caratere");
System.in.read();
```

Account repository

```
public void populateData(){
    for(int i=0;i<10;i++){
        BankAccount bankAccount= BankDirector.accountBuilder()
                .balance(20000+Math.random()*12000)
                .type(Math.random()>0.5? AccountType.SAVING_ACCOUNT:AccountType.CURRENT_ACCOUNT)
                .status(Math.random()>0.5? AccountStatus.CREATED:AccountStatus.ACTIVATED)
                .currency(Math.random()>0.3?"USD":"MAD")
                .build();
        save(bankAccount);
    }
    System.out.println("***************");
    System.out.println(Thread.currentThread().getName());
    System.out.println("Account counter : "+accountsCount);
    System.out.println(bankAccountMap.values().size());
    System.out.println("*****************");
}
```

```
Main
C:\Users\Zakia\.jdks\corretto-17.0.6\bin\java.exe "-jav
Singleton Instatiation
taper caratere
***************
Thread-1
***************
Thread-4
***************
Thread-3
***************
Thread-2
***************
Thread-0
Account counter : 10
Account counter : 40
50
Account counter : 20
*****************
50
*****************
Account counter : 50
50
70
*****************
***************
Thread-7
Account counter : 80
80
*****************
***************
Thread-9
Account counter : 90
90
*****************
***************
Thread-8
Account counter : 100
100
*****************
BankAccount{accountId=93, balance=20104.76675657976, currency='USD', type=CURRENT_ACCOUNT, status=CREATED}
BankAccount{accountId=95, balance=23736.61466032947, currency='USD', type=CURRENT_ACCOUNT, status=ACTIVATED}
BankAccount{accountId=97, balance=21353.56978439044, currency='USD', type=CURRENT_ACCOUNT, status=ACTIVATED}

Process finished with exit code 0
```

Le compteur est atteint 100 tandis que on a juste 97comptes, parce que les threads accèdent presque en même temps au variable compteur sans prendre en considération le temps de communication de la RAM avec UAL de CPU.

Pour cela il faut verrouiller le variable compteur lorsque un thread l'utilise

1- **Synchroniser l'objet :**

```
@Override
public BankAccount save(BankAccount bankAccount) {
    Long accountId;
    synchronized (this){
        accountId=++accountsCount; //critical zone
    }

    bankAccount.setAccountId(accountId);
    bankAccountMap.put(accountId,bankAccount);
    return bankAccount;
}
```

2- **Synchroniser la méthode :**

```
@Override
public synchronized BankAccount save(BankAccount bankAccount) {
    Long accountId=accountId=++accountsCount;
    bankAccount.setAccountId(accountId);
    bankAccountMap.put(accountId,bankAccount);
    return bankAccount;
}
```

3- **Synchroniser la méthode statique getInstance**

```
2 usages
public synchronized static AccountRepositoryImpl getInstance(){
    if(accountRepository==null) {
        System.out.println("Singleton Instatiation");
        accountRepository = new AccountRepositoryImpl();
    }
    return accountRepository;
}
```

6. Implémenter le pattern Prototype pour la classe Compte

C'est du clonage : création de copie

Pour faire , il faut implémenter l'interface cloneable et redéfinir la méthode clone.

```
public class BankAccount implements Cloneable{

    5 usages
    private Long accountId;
    5 usages
@Override
public BankAccount clone() throws CloneNotSupportedException {
    BankAccount bankAccount=(BankAccount) super.clone();
    return bankAccount;
}
```

Faites attention au cas où on a un objet a l'intérieur d'un autre, dans ce cas :

```java
public class Costumer implements Cloneable{
    4 usages
    private Long id;
    4 usages
    private String nom;

@Override
public Costumer clone() throws CloneNotSupportedException {
    return(Costumer) super.clone();
}

@Override
public BankAccount clone() throws CloneNotSupportedException {
    BankAccount bankAccount=(BankAccount) super.clone();
    bankAccount.setCostumer(this.costumer.clone());
    return bankAccount;
}
```

Test :

```java
public static void main(String[] args) throws CloneNotSupportedException {

    BankAccount account1=new BankAccount();
    account1.setAccountId(1L);
    account1.setBalance(40000);
    account1.setCurrency("MAD");
    account1.setType(AccountType.CURRENT_ACCOUNT);
    account1.setStatus(AccountStatus.ACTIVATED);
    account1.setCostumer(new Costumer( id: 1L, nom: "Mohammed"));

    BankAccount account2=account1.clone();

    account1.getCostumer().setNom("Hassan");

    System.out.println(account1.toString());
    System.out.println(account2.toString());

}
```

```
Test
BankAccount{accountId=1, balance=40000.0, currency='MAD', type=CURRENT_ACCOUNT, status=ACTIVATED, costumer=Costumer{id=1, nom='Hassan'}}
BankAccount{accountId=1, balance=40000.0, currency='MAD', type=CURRENT_ACCOUNT, status=ACTIVATED, costumer=Costumer{id=1, nom='Mohammed'}}
```