Laborationsrapport för inlämningsuppgift 2

D0018D, Objektorienterad programmering i Java

Laboration: Inlämningsuppgift 2

**Anders Johansson**

**andjox-1@student.ltu.se**

2022-09-18

# Innehållsförteckning

Inledning 1

Genomförande 1

Systembeskrivning 2

Diskussion 4

# Inledning

Denna laborationsrapport behandlar inlämningsuppgift 2 i kursen D0018D, objektorienterad programmering i Java, som är ett program som hanterar ett simpelt fiktivt banksystem. I programmet hanteras information om bankens kunder och deras konton. I uppgiften finns angivet vilka klasser som ska användas, och även ett antal metoder. Programmets funktionalitet finns också noga beskrivet i uppgiftens instruktioner. Inga klasser utöver de i uppgiftens beskrivning finns i programmet. Däremot har jag lagt till ett antal egna metoder för att koden ska bli mer effektiv.

För en specifik kund i systemet kan programmet hämta uppgifter om personnummer, förnamn, efternamn, vilka konton kunden har, samt alla kontons kontonummer, saldo, räntesats kontotyp, transaktioner med datum och insatt eller uttaget belopp, och även resultatet av räntan när ett konto avslutas. Programmet kan också lägga till nya kunder, lägga till nya konton på en specifik kund, men även ta bort (avsluta) en kunds konto/konton, samt ta bort hela kunder. När en kund tas bort tas även kundens alla konton bort. Det finns också en funktion som ändrar en kunds namn, men personnummer går inte att ändra. Kanske det viktigaste av allt är att man kan sätta in och ta ut pengar från valt konto. Man kan inte ta ut mer pengar än vad som finns på ett konto, och man kan inte sätta in noll kronor eller mindre. Insatt och uttaget belopp är alltid i hela kronor. Sparkontot har ett gratis uttag. Därefter läggs en avgift på 2 % av uttagsbeloppet på. Kreditkontot har 0,5 % sparränta och 7 % skuldränta. Kreditgränsen för kreditkonton är 5000 kr.

Förutom testklassen TestBank2, består programmet av fem klasser; Customer, Account, SavingsAccount, CreditAccount och BankLogic. Account är basklass till SavingsAccount och CreditAccount. Testklassen säkerställer att koden är korrekt, och att alla bankens funktioner fungerar som de ska.

# 

# Genomförande

Eftersom denna inlämningsuppgift bygger på den förra inlämningsuppgiften hade jag det mesta klart för mig redan. Jag la till de nya klasserna och började lägga in den nya koden som krävdes. Jag upptäckte att jag kunde förbättra även den befintliga koden en hel del från förra inlämningsuppgiften, så jag kodade om stora delar av programmet. Jag tycker att det blev mycket bättre. Mer om det senare.

Efter att ha ändrat det mesta som jag tyckte behövde ändras testade jag programmet med testklassen. Eftersom jag ändrade på mycket utav koden så kände jag att det blev ganska rörigt att avlusa koden. Jag gick då igenom klassen BankLogic uppifrån och ner, och ändrade på allt i alla metoder som behövde uppdateras systematiskt i turordning.

# Systembeskrivning

I förra inlämningsuppgiften skickade jag listor med kontonas variabler mellan metoderna. Jag hade metoder i Customer-klassen som hämtade info om konton och returnerade ArrayLists med alla kontonummer, alla kontons saldo etc. I stället gjorde jag i denna versionen en metod i Customer som returnerar ett Account-objekt baserat på vilket indexnummer som skickas in i metoden. En annan metod i Customer returnerar antalet konton i listan med konton. På så sätt kunde jag göra en for-loop med antalet konton, och skicka in ett indexnummer i taget och iterera genom alla konton, och på så sätt komma åt alla kontons alla variabler. Detta sättet var såklart mycket mer effektivt. Eftersom bara ett konto-objekt skickas från Customer-klassen, så skickas heller ingen information i onödan när man behöver information från ett konto.

Nytt för denna version är att transaktioner på kontona sparas i en ArrayList med information om datum och tidpunkt för transaktionerna, samt belopp som satts in eller tagits ut, och saldot på kontot efter transaktionen. Som uppgiftens instruktioner anger, skapade jag en metod BankLogic som returnerar en ArrayList med transaktionerna. Denna metod tar in personnummer på kunden och kontonummer. Metoden söker igenom först listan med kunder för att hitta det angivna personnumret, och sen listan med konton för den kunden. Om kund och konto hittades, returneras en ArrayList med alla transaktioner.

Inuti Accounts skrev jag en metod som heter saveTransaction som tar in en int som representerar beloppet som transaktionen gäller. Först tar jag in datum och klockslag med SimpleTimeFormat i en variabel som heter currentTime:

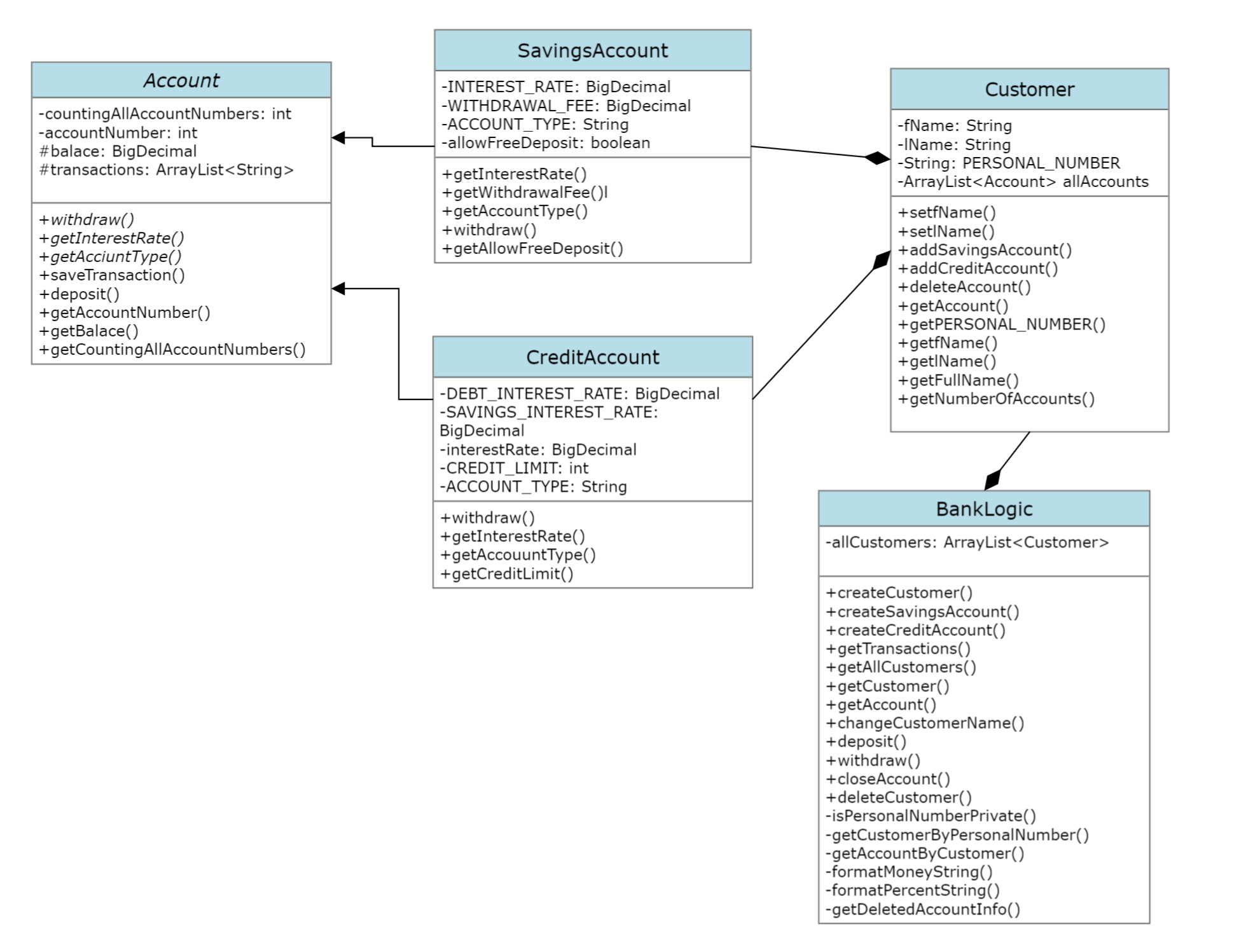
SimpleDateFormat time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
String currentTime = time.format(new Date());

Jag formaterar sen beloppet och saldo till varsin String, och lägger sen in en String sist i array-listen transactions med transaction.add(currentTime + ” ” + formattedAmount + ” Saldo: ” + formattedBalance.

String formattedAmount = NumberFormat.*getCurrencyInstance*(new Locale("sv","SE")).format(amount);  
String formattedBalance = NumberFormat.*getCurrencyInstance*(new Locale("sv", "SE")).format(balance);  
transactions.add(currentTime + " " + formattedAmount + " Saldo: " + formattedBalance);

Programmet struktur består av fem klasser. SavingsAccount och CreditAccount ärver från den abstrakta klassen Account, och dess objekt är members i Customer. Klassen Account är abstrakt eftersom inga objekt (konton) ska instansieras från den. SavingsAccount och CreditAccount kan ses kompositioner till Customer. När en customer tas bort, tas även kontona som hör till kunden bort. Ett konto kan alltså inte existera utan en kund. Konton kan däremot tas bort och läggas till utan att kunden påverkas. En kund har ett konto, ett konto är en del av en kund. Customer-objekt är members av klassen BankLogic, och Customer är komposition till BankLogic.

UML – klassdiagram

-----------------------------

# Diskussion

Den största delen av programmet löstes såklart i förra versionen av banken, i inlämningsuppgift 1. Men eftersom jag ändrade på mycket i denna versionen och försökte förbättra saker, så tycker jag att denna uppgiften blev minst lika omfattande.

Till en början gjorde jag två separata listor för sparkonto och kreditkonto. Vid ett tillfälle kände jag att jag ville skapa en ArrayList med alla konton i samma för att underlätta saker. Sedan läste jag instruktionerna igen och hade missat detaljen att det faktiskt skulle vara en lista med alla konton.

När jag skrev metoden withdraw() i BankLogic, så hämtade jag Account-objektet baserat på personnumret och kontonumret. Men eftersom uttagsavgiften bara gäller sparkontot, så behövde jag kolla om Account-objektet tillhörde SavingsAccount eller CreditAccount. Alltså, jag ville utföra en sak annorlunda på objektet beroende på vilken klass det tillhörde. Jag läste mig till att jag kunde använda getClass() för detta ändamål. Jag använde mig av en boolean allowFreeWithdrawal i SavingsAccount för att kolla om ett gratisuttag redan vad utnyttjat. Eftersom denna variabel fanns endast i SavingsAccount gjorde jag:

/\* If the account is a savings account \*/  
if (account.getClass() == SavingsAccount.class) {  
  
 /\* Check if there is a free deposit on the account \*/  
 if (!((SavingsAccount)account).getAllowFreeWithdrawal()) {  
  
 /\* Add deposit fee \*/  
 amount += SavingsAccount.*getWithdrawalFee*().divide(new BigDecimal("100")).multiply(new BigDecimal(amount)).intValue();  
 }

Jag testade också att göra ACCOUNT\_TYPE till en instansvariabel i Account och sätta varje kontos kontotyp i konstruktorn istället, och på så sätt komma åt kontotypen för varje konto. Men jag bestämde mig för att göra ACCOUNT\_TYPE till en statisk final i både SavingsAccount och CreditAccount, och satte värdet direkt. Eftersom kontotyp-variabeln då är statisk, kan jag inte komma åt kontotypen via konto-objektet. Därför valde jag att använda getClass(). Jag tvekade på hur jag skulle lösa just den detaljen. Jag vet inte om jag gjorde det rätta valet, men det kändes rätt att sätta kontotypen till static final eftersom den variabeln är direkt knuten till just den klassen och ska inte under några omständigheter ändras.

**TRANSACTION**

Såhär löste jag insättningar och uttag, samt sparandet av transaktion-informationen:

BankLogic har metoderna deposit() och withdraw(). Dessa metoder genomför inte transaktioner, utan kollar om giltigt personnummer har angetts, och om kontonumret är giltigt. Metoderna kontrollerar också om ett giltigt belopp har angetts, och withdraw-metoden kontrollerar vilken typ av konto som uttaget ska göras på. Om kontot är ett sparkonto, kontrolleras om ett gratis uttag gäller eller inte, sedan sätts allowFreeWithdrawal till false. När alla kontroller har gjorts anropas metoden makeTransaction(int amount) i Account-klassen. Eftersom withdraw-metoden skickar ett negativt belopp, kan en och samma metod användas i Account för att genomföra transaktioner, samt anropa metoden saveTransaction.

Genom att göra på detta sätt kan saveTransaction hållas privat inom Account. Jag gick över programmet noggrant och undertiden jag skrev rapporten kom jag på denna finurliga lösning, som jag tror att jag är ganska nöjd med. Dels för att jag kunde ta bort en hel del rader kod, och dessutom förbättra inkapslingen genom att göra saveTransaction privat.

Jag tycker att uppgifterna har varit utmanande och jag har lärt mig mycket.

Anders Johansson, andjox-1