Laborationsrapport

D0018D, Objektorienterad programmering i Java

Laboration: Inlämningsuppgift 2

Salim Daoud

saldao-8@student.ltu.se / salim_daoud@hotmail.com

2018-12-02

Innehållsförteckning

Inledning

Uppgiften går ut på att utvidga banksystemets systembeskrivning så att bankens kunder nu ska kunna öppna konton av olika typer; sparkonto respektive kreditkonto.

Genomförande

Jag forsatte arbeta i den integrerade utvecklingsmiljön Eclipse. Eclipse är väldigt kraftfullt som förenklar skapandet av paket och klasser, upprätthålla och bevara relationerna, felsökning m.m. Eclipse har också fiffiga funktioner för att refaktorisera sin kod vilket var aktuellt nu när banksystemet skulle utvidgas med nya och/eller omformade funktioner och klasser.

Jag började mitt arbete med att skapa ett nytt projekt för att sedan kopiera över klasserna från inlämningsuppgift 1 till detta nya projekt. Under utvecklingen så såg jag till att först skapa basklassen och de variabler jag misstänkte var gemensamma och viktiga för att beskriva min modell för konton. Jag funderade på deras tillgängligheter utanför klassen, för att sedan skapa dess metoder. Här var det viktigt att skilja på vad som skall vara tillgängligt inom samma klasshieriarki och vad som skall vara tillgängligt utanför klasshierarkin. Under denna process såg jag återigen till att skapa temporära main klasser för varje ny (eller omformade) klass jag utvecklade där jag stegvis skapade objekt av klassen och testade dess gränssnitt och beteende. Detta för att slutligen när alla nya klasser för systemet var färdigt testas med testklassen vi har blivit försedda med. När jag sedan arbetade med att finslipa och vidareutveckla systemet så såg jag till att ständigt testa hela banksystemet med testklassen för att säkerställa att jag inte har förstört ngt och att systemet fortfarande är intakt och fungerar som det ska.

Jag fortsatte även att arbeta med versionshanteringsverktyget ”Git” och webbhotellet ”Github” för att hålla koll på mina ändringar och lagra min kod i ett säkert ställe.

Kommentering av klasser och metoder har gjorts med hjälp av Javadoc. När det gäller annan kommentering av koden och det som sker där så har jag försökt att praktisera filosofin ”clean code” och därmed koncentrera mig på att koda så strukturerat och tydligt som möjligt så att koden snarare blir självförklarande. Därmed kommenterar jag endast där det kan anses vara behövligt eller otyligt.

För utformandet av UML diagrammet använde jag mig utav mjukvaran Astah som är gratis för studenter.

Systembeskrivning

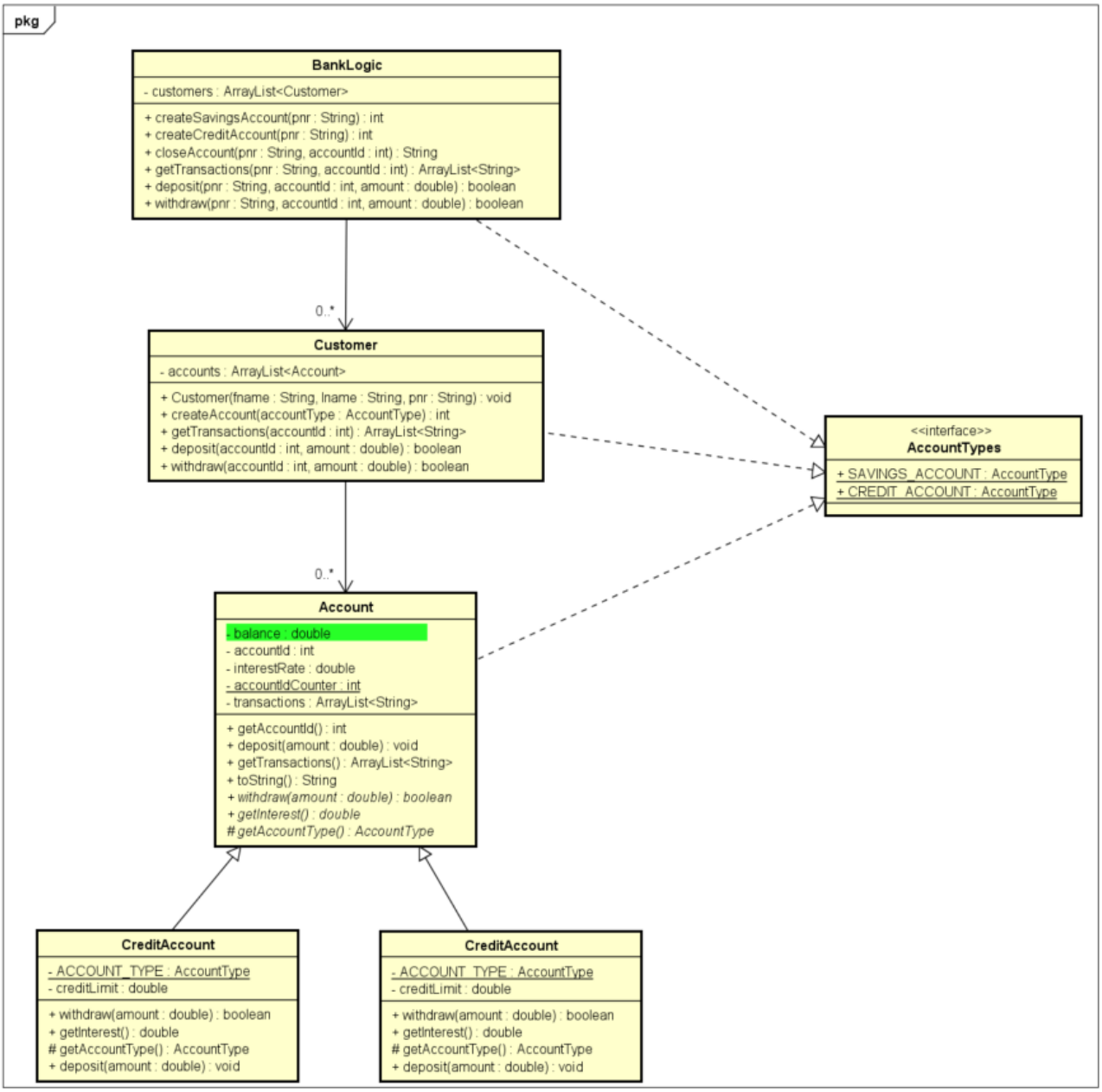
Systemet är utvidgat med tre smarta objekt av tre olika definitionsklasser; en abstrakt klass som representerar vad ett konto är, en klass som representerar ett sparkonto och en klass som representerar ett kreditkonto. Tanken med att skapa en abstrakt klass för ett konto är dels för att skapa en modell av vad ett konto är och beteer sig, och dels för att samla gemensam kod i en basklass som andra subklasser kan återanvända och utvidga. Övriga klasser i banksystemet kan i viss mån endast använda sig utav referenser av basklassen för en del aktioner vilket förenklar koden ytterligare. På detta vis förlitar man sig på Javas inbyggda funktioner för polymorfism, virtuella funktioner och dynamiska bindningar så att rätt subklass anropas och utför de önskade aktionerna. Därtill skapades även ett gränssnitt med konstanter.

SavingsAccount klassen implementerar - likt tidigare - en subklass för sparkonton. Denna klass hanterar all information och tjänster som är relaterade till ett sparkonto som saldo, räntesatser, utföra transaktioner etc.

CreditAccount klassen implementerar en subklass för kreditkonton. Denna klass hanterar all information och tjänster som är relaterade till ett kreditkonto som saldo, kreditgräns, räntesatser, utföra transaktioner etc.

Account klassen implementerar en basklass för konton generellt. Denna klass innehar allt gemensamt utseende och beteende konton har.

AccountTypes är ett gränssnitt som definierar konstanter för olika kontotyper som sedan kan implementeras dvs användas i övriga klasser vid behov.



Diskussion

Account

- En basklass ska definiera ett grundutseende och grundbeteende, den ska innehålla det gemensamma och ska därmed underlätta skapandet av andra subklasser. Mitt mål var att implementera dessa ideér för att skapa en modell av begreppet ”konto” och vad ett konto därmed är. Jag försökte att lägga så mycket gemensam kod som möjligt och rimligt i basklassen så att denna kunde få så stor potential som möjligt. I och med detta

kan jag senare i andra klasser använda referenser av denna klass för att hantera subklassens objekt och aktivera deras metoder. (Galjic 2013, 687-691)

- Jag lade kanske ner mest tid på att fundera på hur jag skulle bygga upp basklassens variabler och vilken åtkomst dessa variabler skulle ha. Subklasserna kommer sedan att ärva dessa variabler – både instansvariabler och statiska variabler (Galjic 2013, 643). ”balance”, ”accountId” och ”transactions” kändes som naturliga instansvariabler, men även ”interestRate” som tidigare (i första versionen) var statisk fick nu nöja sig med att bli en instansvariabel. Anledningen till detta är att räntan nu skulle variera och ändras beroende på kontotyp, men även beroende på saldo och andra villkor. ”accountIdCounter” fick fortsätta vara statisk då varje konto – oberoende kontotyp – ska ha ett unikt id nummer. Det intressanta föll på den tidigare statiska variabeln för kontotyp. I och med att jag tyckte att en kontotyp är karaktäristisk för ett konto så ville jag ha med denna i basklassen. Dock kunde denna variabel då inte vara statisk i och med att detta skulle innebära att det sista satta värdet (vid skapandet av en subklass) skulle vara det som gäller för alla objekt oavsett kontotyp. Inte heller ville jag att denna variabel skulle bli en instansvariabel då kontotypen var gemensam för alla objekt av samma subklasstyp. Jag tyckte att den bästa medelvägen och lösningen – inspirerad av Bozhanov (”static field in abstract class”, 2011) – var att behålla denna statiska variabel, men att ”flytta ned” denna till respektive definitionsklass av subklasserna och att sedan i basklassen deklarera en abstrakt ”get”-metod (getAccountType) som kan aktiveras av en referens av basklassen men som definieras i subklasserna för att förse intressenten med vilken kontotyp den tillhör.
- I och med att denna klass inte har så många variabler och att dessa i subklassern inte ingår i några komplicerade och invecklade algoritmer och funktioner som kan göra koden svårläst så valde jag att deklarera alla dessa som privata variabler. I vissa fall rekommenderas det att deklarera variabler i en basklass som ”protected” då de behöver vara åtkomliga och ska användas i subklasserna men nackdelen med detta är att fler klasser får möjlighet att manipulera dessa – subklasser och alla andra klasser som tillhör samma paket – vilket innebär en mindre säkerhet, kontroll och att det är lättare att göra fel. Däremot deklarerades hjälpmetoder och ”get/set”-metoder som protected – snarare än publika – för att underlätta implementeringen i subklasserna och då dessa metoder inte heller är tänkta att användas av andra ”utomstående” klasser. (Galjic 2013, 659)
- I och med att basklassen är abstrakt så kommer det aldrig gå att skapa ett objekt av denna klass (Galjic 2013, 693). Därmed så utelämnade jag initieringen (i konstruktorn) av de variabler som har värden som är specifikt bestämda beroende på vilken subklass den tillhör. Tyckte det var ”fult” att tilldela värden som sedan bara kommer att ersättas i subklasserna och att det är missvisande. Jag tilldelade i och med detta endast de variabler som hade naturliga och logiska värden som även överenstämde med det önskade utgångsvärde man vill ha.
- Denna klass innehåller även en del abstrakta metoder. Tanken med dessa metoder är att deras implementationer är specifika för deras subklasser, men att dessa samtidigt behövs i klassen då de är grundläggande egenskaper för ett konto. Genom att deklarera dessa som abstrakta så överlåter man den konkreta implementeringen av dessa metoder till respektive subklass men att de samtidigt ingår i modellen av

begreppet ”konto” och går att aktivera via en basklassreferens. I och med detta har man även skapat en vettig startpunkt i klasshierarkin. (Galjic 2013, 690 - 692)

- Jag skapade ett gränssnitt. Största anledningen till detta var att jag ville ha en mer generisk metod för skapandet av konton av olika kontotyper i klassen ”BankLogic” och därmed ville jag skapa en uppsättning konstanter för varje kontotyp (mer om detta under ”Customers” diskussionspunkter). Jag upptäckte senare att det vore intressant och bra att återanvända gränssnittets konstanter för subklassernas statiska variabel för kontotyp. I och med detta var denna definitionsklass tvungen att ”implementera” eller snarare deklarera detta gränssnitt då ”get”-metoden för kontotyp returnerar gränssnittets datatyp. (Galjic 2013, 762)
- En ny funktionalitet som skulle läggas till var att logga alla transaktioner som sker. I och med att detta är en gemensam funktionalitet för alla subklasser och ett utmärkande drag för ett konto, så placerades detta i basklassen. I samband med detta skapades en hjälpfunktion som skapar ett transaktionsdatum och som deklarerades med åtkomstattributet ”protected” iom att denna hjälpmetod även behövs vid implementeringen av den abstrakta metoden ”withdraw” i subklasserna. Den lilla utmaningen med denna funktionalitet var att få till det specificerade formatet för datum och tid. Fann ett litet tips av benmessaoud (”Convert java.util.Date to String”, 2011) som utgick ifrån att använda en formaterare av typen ”SimpleDateFormat” och med hjälp av denna sätta önskat format för att sedan skapa en sträng representation av datumet i just det formatet.

SavingsAccount

- Jag har redan redogjort för anledningen bakom användandet av gränssnittet ”AccountTypes” för subklassernas variabel för kontotyp. Dock värt att nämna är att detta gränssnitt ärvs av sin basklass och behöver därmed inte deklareras igen i denna definitionsklass. Vidare så deklarerades denna variabel som en statisk konstant i och med att den är en klass resurs snarare än en instans resurs. Kontotypen bör heller inte ändras för klassen vilket påverkade deklarationen till att inkludera ”final” för konstant.
- Enligt Galjic (”Implementera konstruktorer i en subklass”, 2003, 653) så anropas basklassens förvalda konstruktur (via super anropet) automatiskt om detta inte anges som första sats i subklassens konstruktur. Detta är ytterligare en anledning till att skapa en förvald konstruktor (dvs konstruktor som saknar parametrar) i basklassen för att förhindra kompileringsfel. Trots detta anser jag att det är bättre att vara tydlig och säkrare att ange super-anropet för att anropa basklassens konstruktor. Det är viktigt att detta sker korrekt i och med att basklassen initierar viktiga instansvariabler som konto id och saldo. Även viktigt att super-anropet sker först så att man t.ex. inte råkar initiera eller utföra några som helst aktioner i subklassen som baseras på basklassens oinitierade variabler och tillstånd. Kan även tillägga att denna strategi även tydliggör vad som är gemensamt och vad som är specifikt för subklassen då endast initiering av subklassens specifika variabler explicit görs i subklassens konstruktor.
- Klassen implementerar endast basklassens abstrakta metoder i och med att dessa är de metoder vars implementation är specifika för denna subklass. Värt att nämna är att även denna klass skulle bli abstrakt och behövas att deklareras som abstrakt om jag skulle missa att implementera någon av basklassens abstrakta metoder. I detta fall

med detta lilla system och lilla klasshierarki, så var det tydligt vad som behövdes göras. (Galjic 2013, 693)

- En annan fundering jag stötte på var hur jag skulle kommentera (via Javadoc) de omdefinierade metoderna i subklassen. Detta i och med att de redan hade dokumenterats i basklassen. Jag fann då i Oracles Java dokumentation att subklassens metod (om inte kommenterad) automatiskt ärver kommentarerna från sin basklass respektive metod som har blivit omdefinierad. Man kan även explicit uppnå detta med användningen av javadoc taggen ”{@inheritDoc}”. Jag valde att hålla mig till Eclipse inbyggda lösning på detta som genererar en sk. ”icke”-javadoc kommentar som hänvisar till basklassens dokumentation av metoden i och med att detta var tydligare än att inte ha någon kommentering alls.

CreditAccount:

- Jag funderade på om jag behövde deklarera variablen för kreditgränsen till statisk eller ej. Beslutet att inte ha denna som statisk föll på att jag ansåg att krediten kan skilja sig mellan olika kunder i och med olika ekonomiska tillstånd och styrkor. Därmed fick denna bli en instansvariabel; specifik och förändringsbar för varje kund.
- En annan intressant detalj i denna subklass är metoden ”deposit”. Denna metod har ärvts från basklassen men omdefinieras i denna subklass. Observera även att basklassens deklaration för denna metod inte är abstrakt. Anledningen till detta är att metoden är karaktäristisk för ett konto och att jag ansåg att båda subklasserna i grunden utgår från samma funktionalitet (även möjliga framtida subklasser). Därmed kunde själva grunfunktionaliteten definieras i basklassen och sedan återanvändas fullt ut i ”SavingsAccounts” klassen och i denna klass. Jag valde därmed att först anropa basklassens implementering av metoden genom super-anropet för att sedan lägga till den lilla specifika koden för just denna klass (kontrollen om räntan behöver ändras). Detta är en teknik som används ofta där man i den omdefinierade metoden i subklassen utgår från basklassens variant av metoden och anpassar denna funktionalitet efter sina egna önskemål. (Galjic 2013, 657)

Customer

- Som tidigare nämnt öppnade användningen av en basklass upp i sin tur för förenklingen av kod genom användandet av referenser av basklassen snarare än specifika referenser för respektive subklasstyp. Denna klass kan nu tillhandahålla basklassreferenser för konton och aktivera alla nödvändiga aktioner på denna referens utan att behöva göra specifika listor/variabler och kontroller för att först ta reda på vilken typ av konto ett konto nummer tillhör för att sedan göra sina önskade aktioner. Koden blir därmed bl.a. mindre, renare och tydligare. Jag kan därmed förlita mig på Javas mekanismer för polymorfiska anrop och dynamiska bindningar, där basklassreferensen är en smart referens som kommer att se till att välja rätt variant av omdefinierad metod beroende på det refererade objektets typ. (Galjic 2013, 670-671)
- Gällandes skapandet av konton av olika kontotyper så kändes det renare och bättre att tillhandaha endast en metod för detta och som därmed behövde deklareras mer generisk. För att uppnå detta behövde jag ha en parameter i metoden som avgjorde

vilken typ av konto man vill skapa. Därmed blir det en enda metod men med en parameter extra. Detta är även bra och mer logisk om fler kontotyper skall användas i framtiden snarare än att ha en uppsättning av specifika metoder för varje kontotyp med liknande kod. I och med detta ville jag skapa en uppsättning konstanter för varje kontotyp som kunde användas av metoden. Resultatet blev ett interface ”AccountTypes” som sedan ”implementerades” dvs ”utpekades” eller deklarerades av de definitionsklasser som är intresserade av denna. Användingen av datatypen ”enum” kändes naturlig då det är mer lättläst och enklare att hantera. (Galjic 2013, 762)

BankLogic:

- Vad gäller denna klass så var förändringen inte större än att utvidga klassen med ytterligare två mer eller mindre rättframma metoder. Förändringarna med att införa en basklass för konton påverkade ej denna klass i och med att denna klass inte har några referenser till några sin helst konton utan endast till kunder.
- Jag funderade dock på kravspecifikationen som nämner att en del information ska visas på skärmen när ett konto avslutas. Jag tolkade detta som att man endast behöver skriva informationen via Javas metod ”System.out.println”.

AccountTypes

- Jag valde att använda mig utav enum istället för byte eller int eller liknande. Detta då det är enklare och att jag egentligen inte bryr mig om vilket värde de har då de ska användas som konstanter som representerar en typ.
- Jag deklarerade inte konstanterna som ”public static final” i och med att detta är underförstått i Java då det handlar om konstanter i ett gränssnitt. (Galjic 2013, 762)

Referenser

Galjic, Fadil. 2013. *Programmeringsprinciper i Java*. 1:2. Lund: Studentlitteratur AB.

Bozhanov, Bozhidar. 2011. *“Static field in abstract class”*. [Elektronisk]. Tillgänglig: https://stackoverflow.com/a/4898769 [2018-12-03].

benmessaoud, Ali. 2011. *“Convert java.util.Date to String”*. [Elektronisk]. Tillgänglig: https://stackoverflow.com/a/5683761 [2018-12-04].

Oracle Java SE Documentation (u.å). *” javadoc - The Java API Documentation Generator”*. [Elektronisk]. Tillgänglig: https://docs.oracle.com/javase/6/docs/technotes/tools/solaris/javadoc.html#inheritingcomments [2018-12-04].