14-12-2018

KEA Bank

1. Semester projekt

Julian Christensen, Patricia Kellvig,
Knud Billing, Johannes Linde-Hansen,
Thomas Dahl og Martin Wagner Busk

Indholdsfortegnelse

Introduktion

Denne rapport er udarbejdet af Patricia Kellvig, Knud Billing, Thomas Dahl, Johannes Linde-Hansen, Julian N. Christensen og Martin Wagner Busk i forbindelse med 1. semester projektet i fagene ”Systemkonstruktion”, ”Systemudvikling” og “IT-Organisationer”.

Projektet gik ud på at udarbejde et banksystem der skulle kunne forbinde et java program med en MySQL database. Systemet skal, blandt andet, kunne håndtere oprettelse af kunder og forskellige typer af konti samt håndtere transaktioner - både internt og ud af banken.

I denne rapport har vi dokumenteret vores ideer og overvejelser om de givne krav til projektet, konstruktionen af systemet, hvordan databasen er sat op, hvordan vi har brugt SCRUM og hvilke forretningsmæssige overvejelser vi har gjort.

“Intet program er helt færdigt før det kan sende mails.”

- Morten Voetmann Christiansen

Status på projektet

Alle givne user stories er håndteret, og systemet går korrekt igennem alle opstillede tests. Alle opståede særtilfælde - eksempelvis med overførsler af negative beløb eller ved oprettelse af eksisterende kunder - er håndteret, og resulterer i korrekt tilbagemelding til brugeren.

Skulle systemet udvikles videre er der flere punkter, hvor det ville være naturligt at udfylde nogle huller i systemet.

På nuværende tidspunkt er flere værdier håndteret som konstanter - især gebyrer ved overførsler og ved overtræk skal håndteres mere dynamisk. Rentesatser er desuden bare værdier - der bliver aldrig håndteret nogen former for afkast.

Sikkerhed har ikke været et fokuspunkt under projektet, og er derfor ikke håndteret. Banker i Danmark ville naturligt benytte NemID til at sikre at kundernes oplysninger kun er tilgængelige for sikkerhedsgodkendt personale og kunden selv. Andre former for sikkerhedscheck skal implementeres for at sikre at kun administratorer har administratorrettigheder osv.

Som systemet fungerer nu, foregår al kommunikation i terminalen - her ville det være oplagt at udvide til web-adgang, Android/IOS applikationer o.lign. Dette vil selvfølgelig også kræve at der kommunikeres med andet end en server lokalt hos brugeren.

Desuden ville det være naturligt at udvide til at kunne understøtte samme typer tjenester som andre banksystemer tillader. Funktioner som at kunne spærre konti, tilbyde forskellige typer lån og finansiering, samt effektiv kommunikation mellem banken og kunderne ved chatsystem/mails/sms/telefon.

Brugervejledning

Inden programmet kan køres på brugerens system, da skal brugeren oprette et login med tilstrækkelige privilegier til vedkommendes MySQL database med følgende oplysninger:

Hostname: localhost
Port: 3306
Username: user
Password: 1234

(Dette kan rettes i filen Main.java på linjerne 3-7)

Når programmet køres, præsenteres brugeren med en ramme konstrueret af ASCII-karakterer, hvori mulige funktioner står listet sammen med instruktioner for at navigere mellem disse. Navigation foregår ved at indtaste den kommando der står beskrevet i rammen, og afslutte med "Enter". (Eksempelvis: <Tast "1" for … > <Afslut med "Enter">). På langt størstedelen af siderne vil desuden være muligheder for at taste "B" for at gå et skridt baglæns, "M" for at gå til hovedmenuen, og "Q" for at lukke programmet.

I den første ramme brugeren præsenteres for vælger brugeren hvilken type bruger de er - valget står mellem "kunde", "medarbejder" og "administrator".

Hvis bruger er en kunde, da vi de have mulighed for at overføre penge eller få vist alle deres konti og oplysninger for disse.

Er brugeren medarbejder forefindes flere andre muligheder - Oprettelse af nye kunder eller fremsøgning af eksisterende kunder, hvorefter valgte kunders konti kan modificeres på forskellig vis.

Som administrator kan brugeren få vist bankens status samt nulstille, genindlæse og tage backup af banken.

Kravspecifikation

Kravliste

Til projektet er sat det mål at få alle kundens ønsker, listet herunder, implementeret for således at kunne levere et velfungerende system til banken.

Product backlog:

1. As a bank employee, I want to add a new customer to the system.
2. As a bank employee, I want to create a savings account for a customer.
3. As a bank employee, I want to create a current account for a customer.
4. As a bank employee, I want to allow compound interest to a savings account for a customer.
5. As a bank administrator, I want to create a facility report of any given account's status.
6. As a customer, I want to deposit money into my savings account.
7. As a customer, I want to deposit money into my current account.
8. As a customer, I want to withdraw money from my savings account.
9. As a customer, I want to withdraw money from my current account.
10. As a customer, I want to make transfers with my savings account.
11. As a customer, I want to make transfers with my current account.

For at sikre at disse funktioner er implementeret korrekt skal systemet igennem en række tests.

Test-cases:

1. Overførsel af gyldigt beløb fra opsparingskonto til egen lønkonto.
2. Overførsel af gyldigt beløb fra opsparingskonto til egen opsparingskonto.
3. Overførsel af for stort beløb fra opsparingskonto til egen lønkonto.
4. Overførsel af negativt beløb fra opsparingskonto til egen lønkonto.
5. Overførsel af gyldigt beløb fra opsparingskonto til anden kundes lønkonto.
6. Overførsel af gyldigt beløb fra opsparingskonto til ugyldig konto.
7. Overførsel af gyldigt beløb uden overtræk fra lønkonto til egen opsparingskonto
8. Overførsel af gyldigt beløb med overtræk fra lønkonto til egen opsparingskonto.
9. Overførsel af negativt beløb fra lønkonto til egen opsparingskonto.

10. Overførsel af gyldigt beløb uden overtræk fra lønkonto til anden kundes lønkonto i andet pengeinstitut.
11. Overførsel af gyldigt beløb med overtræk fra lønkonto til anden kundes lønkonto i andet pengeinstitut.
12. Overførsel af negativt beløb fra lønkonto til anden kundes lønkonto i andet pengeinstitut.
13. Overførsel af gyldigt beløb uden overtræk fra lønkonto til anden kundes lønkonto i samme pengeinstitut.
14. Overførsel af gyldigt beløb med overtræk fra lønkonto til anden kundes lønkonto i samme pengeinstitut.
15. Overførsel af negativt beløb fra lønkonto til anden kundes lønkonto i samme pengeinstitut.
16. Overførsel af gyldigt beløb uden overtræk fra lønkonto til anden kundes opsparingskonto i samme pengeinstitut.
17. Overførsel af gyldigt beløb med overtræk fra lønkonto til anden kundes opsparingskonto i samme pengeinstitut.
18. Overførsel af negativt beløb fra lønkonto til anden kundes opsparingskonto i samme pengeinstitut.

Konstruktionskrav

1. Der skal være pæne fejlbeskeder ved exceptions.
2. Der skal være pæne fejlbeskeder ved fejlslagne pengeoverførsler.

Reliability-krav

1. Der skal kunne tages daglige backups.
2. Alle ændringer skal straks gemmes i databasen.

GUI:

1. Der skal være en GUI hvor kunden kan se kontooversigt med aktuelt indestående.
2. Der skal være en GUI hvor kunden kan overføre penge mellem egne konti.
3. Der skal være en GUI hvor kunden kan foretage overførsler til eksterne konti fra sine lønkonto.
4. Der skal være en GUI hvor kunden for en konto kan se: reg.nr. konto nr. kontotype, indestående, rentesats, tilladt overtræk.
5. Der skal være en GUI hvor banken kan oprette en ny kunde.
6. Der skal være en GUI hvor banken kan oprette en ny konto.

7. Der skal være en GUI hvor banken kan ændre rentesats på en konto.
8. Der skal være en GUI hvor banken kan ændre tilladt overtræk på en konto.
9. Der skal være en GUI hvor banken kan søge efter kunde på fuldt navn og se alle kontooplysninger for kunden.
10. Banken skal kunne sætte kontanter ind på en kundes konto.
11. Banken skal kunne hæve kontanter fra en kundes konto.
12. Som medarbejder skal man kunne søge efter en bestemt kunde os se kundens oplysninger.

Test

1. Der skal oprettes minimum 10 konti.
2. Der skal oprettes minimum 5 kunder.
3. Til test skal benyttes reg.nr. 9800 og 9900.

Business rules

1. Alle beløb skal være positive - negative beløb skal afvises.
2. Der skal være en rentesats på kontoen.
3. Fra opsparingskontoen må man kun overføre til egne konti.
4. Der skal opkræves overtræksgebyr hvis man er mere i minus end tilladt.
5. Der skal være en kontrol af om modtagerkontoen eksisterer/er gyldig.
6. Hvis man overfører til en konto i en anden bank, skal der opkræves et overførselsgebyr.
7. En opsparingskonto må aldrig gå i minus.
8. Bankens kontantbeholdning må aldrig gå i minus (HUSK EN POSITIV KONTANTBEHOLDNING ER ET NEGATIVT TAL PÅ DENNE KONTO).

Konstruktion

Repræsentation af beløb

For ikke at løbe ind i problemer med tilfældig afrunding af beløb, benyttes ikke double, men long til at repræsentere beløb. Beløb regnes i ører, således at 12,34 kr. i programmet vil være en long med værdien 1234.

Repræsentation af kontonumre

Kontonumre er i programmet repræsenteret ved Strings. De har en fast længde, kan have foranstillede nuller og bruges ikke til matematiske udregninger.

Transaktioner

Der bruges transaktioner (Transaction) for at flytte penge fra en konto til en anden. En transaktion består af to kontonumre og et beløb. Beløbet trækkes fra den ene konto og lægges til den anden. Transaktionen er "atomar", dvs. den enten slet ikke bliver udført, eller at den bliver udført i sin helhed. Der vil på den måde aldrig opstå en situation, hvor der bliver hævet penge på én konto uden at beløbet bliver sat ind på en anden.

Indestående på konti

Vi har baseret vores konti (Account) på idéen om dobbelt bogholderi. For at undgå to værdier (debet og kredit) for hver konto er der kun ét tal der viser kredit minus debet. På den måde vil alle passiver være positive og alle aktiver vil være negative. Det vil sige at alle indlånskonti har positiv værdi når der er penge på dem. Den eneste konto med en negativ værdi når der er penge på den er den konto vi bruger til kontantbeholdningen. Kontantbeholdningen er et aktiv - derfor en negativ værdi.

Overførsel til andre banker

Overførsel til andre banker er implementeret som en konto i banken. Man kan sige at det er "andre bankers" konto i vores bank. Ved overførsler til andre banker undersøges først (ved hjælp af BankRegister-klassen) om kontoen eksisterer. Derefter overføres beløbet til InterBank-kontoen med en String reference der fortæller hvilken konto i den fremmede bank beløbet skal indsættes på.

Dependency injection

Der er lavet et interface for at forbinde til databasen. På den måde er databasen afkoblet fra resten af programmet. Hvis man ønsker at bruge PostgreSQL eller en anden database kan man programmere mod interfacet (Persistence) og det eneste man skal gøre for at bruge den nye kode, er at ændre en enkelt linje (linje 16) i Main-klassen.

Brug af databasen

Databasen bruges i programmet som en art skygge-kopi af banken. Ved programmets start hentes alle oplysninger om banken fra databasen og der dannes passende objekter med tilsvarende værdier. Når data ændres i programmet, skrives de straks til databasen sådan at værdierne i databasen altid stemmer overens med værdierne i programmet.

Hjælpeklasse

Klassen AccountNumber indeholder metoder til at manipulere kontonumre og undersøge om de er formateret rigtigt og er gyldige.

User Interface

Der er valgt en UI løsning med in-/output i en terminal. Denne simple løsning blev valgt fordi en løsning med Swing eller JavaFX ville indeholde for mange ubekendte til at gruppen havde en idé om programmerings-omfanget.

SCRUM

Under udviklingen af projektet er der taget udgangspunkt i SCRUM som udviklingsmetodik. SCRUM er ikke fulgt nøjagtigt, men er blevet tillempet gruppens foretrukne arbejdsform. Når det har været nyttigt, er en SCRUM master blevet udpeget, og denne har da haft projektstyringen, og været boldholder for igangværende opgaver og projektets progression.
Dagligt er der lavet små notater til, hvad der er blevet arbejdet på, snakket om, og hvad der evt. har været af problemer. Der er gjort forsøg med burndown charts, men når der i ugens løb er tilføjet flere ting til sprintens backlog, da har oplevelsen været, at grafen ikke har kunne afspejle dette ordentligt. Et eksempel på dette fra sprint 2 findes i bilag 1.

SCRUM Roles:

Product owner: Douglas og Morten

SCRUM master: skiftende så alle får prøvet det

Developers: alle på nær SCRUM master

Daily SCRUM:

26 november. Mandag. Dag 1:

- Vi har afklaret nogle spørgsmål til lærerne/product owner, for at være sikker på opgaven omkring banken og dens egen konto og i forhold til SCRUM delen.
- Vi har gennemgået planen, med Sprint og rollerne, hvorefter vi i fællesskab diskuterede og derved kom frem til at definere klasser med attributter, så vi alle var med på overordnede.
- Startet på at implementere vores klasser i Java.
- Patricia har meldt sig som SCRUM master. Hun var SCRUM master hele ugen

27 november. Tirsdag. Dag 2:

- Vi fortsatte fra i mandags, med at oprette og implementere klasserne.

28 november. Onsdag. Dag 3:

- Hjemmearbejde, hvor vi hver især kiggede lidt på rapport og lidt programmering.

29 november. Torsdag. Dag 4:

- Vi har i dag forsøgt med mob programmering, så vi alle bedre kunne få overblik over hvad vi lavede og kunne hjælpe hinanden.
- Vi har måtte tilføje flere user stories til vores sprint, da vi har nået de første, så derfor går vores kurve lidt op, på vores burndown Chart.
- Vi har oprettet en ny klasse, hvorfra vi kunne teste systemet.

30 november. Fredag. Dag 5:

- Fortsættelse af mob programming, hvor vi har kørt test af de resterende klasser, og de bestod alle sammen.
- Vi fik løst alle user stories fra product backlog, som gjorde vi kunne få samlet de andre krav fra opgaven ét sted.
- Til slut planlagde vi, hvordan næste uge skulle forløbe og hvad der skulle nås.

1 - 2 december. Weekend (dag 6 og 7):

- Kigget på rapport og lidt programmering.
- Knud har lavet et oprettet et nyt trello board samt defineret flere punkter (user stories/hvad det nu hedder)

3 december. Mandag. Dag 8:

- Start på ny sprint. Sprint backlog udarbejdes
- Knud meldte sig til SCRUM master for dag 8 og 9
- Opsamling på hvad der er blevet lavet indtil nu. Så alle forhåbentlig har en god ide om hvordan tingene hænger sammen
- Der kodes individuelt men med samarbejde

4 december. Tirsdag. Dag 9:

- Test hver især
- Test af hinandens test, så vi er sikre på at de fungere korrekt.

5 december. Onsdag. Dag 10:

- Hjemmearbejdsdag, hvor der er kigget lidt på rapport og lidt på programmering.

6 december. Torsdag. Dag 11:

- Vi har kørt de sidste test, og vi har hjulpet hinanden med problemer.
- Johannes melder sig som SCRUM master i dag og i morgen.

7 december. Fredag. Dag 12:

- Arbejde på rapport.
- Færdiggørelse af test og test af hinandens tests.
- Arbejde på GUI

8 - 9 december. Weekend (dag 13 og 14):

- Vi vil holde weekend, men kigge lidt på rapport og arbejde lidt på GUI hvis vi får tid.

10 december. Mandag. Dag 15:

- SCRUM møde om morgenen, ingen udfordringer, så vi arbejder på GUI og arbejde på forretningsmæssige overvejelser og SCRUM delen i rapporten.
- Patricia og Knud har delt SCRUM master rollen i denne uge, da der har været noget sygdom.

11 december. Tirsdag. Dag 16:

- Vi har arbejdet færdig med GUI delen og arbejdet på rapporten.

12. december. Onsdag dag 17:

- Hjemmearbejdsdag, hvor vi har kigget på rapporten.

13. december. Torsdag, dag 18:

- Der er blevet arbejdet med rapport.

14. december. Fredag, dag 19:

- Rapport skrivning og aflevering
- Der er lavet julepynt, spist æbleskiver og hygget i gruppen
- Glædelig Jul!

Forretningsmæssige overvejelser

Idet nutidens banksystemer er enormt store og avancerede vil et projekt af dette omfang helt naturligt ikke kunne få et ben til jorden i virkelighedens verden. Her behandles alting lokalt, sikkerhed er ikke eksisterende, og interfacet er terminal-baseret. Kort sagt - projektets omfang er alt for småt til at være fungerende i virkelighedens verden.

Der har været en del intern debat om hvorvidt det ville give mening at betragte projektet ud fra IT-billedet i 1960-70'erne. Under den tid gik bankerne fra fysisk bogføring til at få EDB programmer til at effektivisere bankvirksomheden. Dengang ville projektet her kunne effektivisere bankens arbejde ved at de kunne overgå fra manuel til digital bogføring. Dette ville medføre alle de arketypiske fordele historien har vist ved overgange til IT - øget effektivitet, større kapacitet, bedre overblik, færre menneskelige fejl. Denne type teknologi var dog ikke velkendt dengang, og ville, uden tvivl, medføre en del problematiske situationer - hvorfor en "konsulent" fra udviklingsteamet nok burde tilknyttes banken.

Database struktur og design

Entiteterne i databasen er baseret på objekterne i Java programmet. Både fordi databasen blot anvendes som en slags backup og fordi tabeller fungerer på en anden måde end objekter er attributterne dog lidt anderledes.

Mængden af data i den er reduceret til et minimum og redundans er forsøgt fjernet. F.eks. er kontonumre implementeret med 10 tegn i databasen mod 14 tegn i Java-delen.
Konti har ingen balance-attribut i databasen, idet værdien for denne kan beregnes ved summen af transaktioner udført på den pågældende konto.

Alle konti har en fremmed nøgle der angiver hvilken kunde der ejer den - undtagen bankens tre specielle konti (egen konto, kontantbeholdning og interbankkonto) der ikke er tilknyttet nogen kunde og i praksis har CustomerID sat til nul.

Se bilag 3 for E/R diagram.

GUI overvejelser

Grundet manglende erfaring indenfor arbejde med Java Swing eller JavaFX blev det besluttet at holde user interfacet i terminalen fremfor i separat vindue. Dermed blev input begrænset til indtastning, og med en information ad gangen. Ved denne tilgang skal alle valg træffes ved separate indtastninger, hvilket medfører at brugeren bliver præsenteret for en del forskellige skærmbilleder - disse er dog ganske simple, og vil derfor ikke fremstå så overvældende som deres antal kunne antyde.

Til opbygningen af de enkelte skærmbilleder, stod det hurtigt klart at en del elementer skulle gå igen. Overskriften var altid bankens navn, rammebredden konstant, bunden var altid <Godkend med “Enter”>, osv. Det at så mange elementer gentages, medfører at brugeren får en oplevelse af kontinuitet skærmbillederne imellem. Samtidigt resulterede de mange gentagelser i at programmeringsarbejdet blev lettere da meget kode kunne genbruges.

Det blev desuden debatteret hvorvidt skærmbilledernes rammer skulle have konstant højde. Dette blev dog fravalgt, idet arbejdsbyrden ville forstørres, og brugerens terminal alligevel er rullende og ikke kan ryddes mellem nye print.

Til nogle skærmbilleder, hvor brugerens input skal behandles, benyttes samme skærmbillede med tilføjede linjer med fejlmeddelelser, hvis input er ugyldigt. Dette er dog ikke en praksis der er holdt igennem hele programmet, idet det viste sig at være et større arbejde end det tilføjede programmet i værdi.

For at holde overblik over de mange forskellige skærmbilleder til programmets funktioner blev et tillempet workflow-diagram (se bilag 2 - GUI labyrint) udarbejdet. Dette diagram hjalp betragteligt til at holde styr på hvordan de forskellige funktioner interagerede, og hvilke værdier det var relevant at returnere hvortil.

Test

Teststrategi

I forbindelse med software-konstruktionen blev JUnit4 valgt som den foretrukne testmetode. Når først testen er programmeret vil JUnit give svar på testen blot ved at trykke på en knap. Al opsætning og oprydning kan klares af JUnit.

For at sikre at SCRUM-tasks blev udført korrekt, blev alle tasks på SCRUM-board'et testet af et andet medlem af gruppen før de blev flyttet til "færdig"-kolonnen.

Kvalitetstest af brugergrænsefladen blev foretaget manuelt, og uden struktur - det skete efterhånden som udviklingen fandt sted.

Obligatoriske test

Alle de obligatoriske test er implementeret som JUnit4 test. Hvert gruppemedlem har programmeret 3 test som ligger i en .java fil med navnet på gruppemedlemmet + "Test". TestSuite.java samler alle testene så de kan udføres fra en enkelt fil.

Test af brugergrænseflade

Brugergrænsefladen blev modelleret i en tekstfil (gui_model.txt) med billeder af hvert skærmbillede. For at teste om brugergrænsefladen var sammenhængende tegnede vi en graf (bilag 2) over skærmbillederne. Det viste sig at være et godt værktøj til at holde overblik over brugergrænsefladen. Grafen viste at der manglede skærme til oprettelse af nye kunder - en mangel som derefter var simpel at rette.

Hvad angår brugervenligheden af grænsefladen, blev dette "testet" af gruppens medlemmer efterhånden som programmet blev udviklet.

Dette har umiddelbart resulteret i en let forståelig brugergrænseflade, hvor der er en klar følelse af kontinuitet siderne imellem. Skulle dette testes dybere, da skulle eksterne testere sættes til at navigere i systemet.

User stories som tests

Til hver af de givne user stories er skrevet en JUnit4 test (Se UserStoryTest.java). Således blev det simpelt at efterprøve hvorvidt disse blev overholdt.

Test driven development

Der blev afprøvet en anelse - og meget uformel - TDD (Test Driven Development). Enkelte tests blev skrevet før funktionaliteten var implementeret. Derefter var målet at bestå testen ved at føje funktionalitet til programmet.

Konklusion på tests

Testing ved brug af JUnit4 var effektivt til at teste at systemets funktionalitet fungerede. Dog var der nye udfordringer ved at skulle sikre at tests rent faktisk testede for den ønskede funktionalitet. Da JUnit4 var kommet tilstrækkeligt ind under neglene, var dette dog ganske ligetil.

De manuelle tests af brugergrænsefladen kunne være mere udførlige - dog lægger projektets omfang ikke rigtigt op til at der skal yderligere kvalitetssikring til.

Class diagram

Vores tanker bag systemet var at det skulle holdes så simpelt som overhovedet muligt, hvilket også afspejles af klassediagram, se bilag 4. Ved brug af denne simple struktur bliver det så enkelt som muligt for bankens ansatte og kunderne at bruge systemet.

"Bank"-klassen er samlingspunkt for en stor del af selve strukturen i systemet. Den indeholder lister over alle kunder, konti og transaktioner. Desuden holdes der styr på de særlige kontonumre, der er tilknyttet bankens interne funktioner; kontantbeholdning, bankens egen konto, og en konto til at håndtere transaktioner ud af banken. "Bank"-klassen har metoder til at oprette nye objekter af de andre tilknyttede klasser; kunder, konti og transaktioner.

"Customer"-klassen håndterer kundedata samt hvilke konti hver kunde har.

"Account"-klassen er en abstrakt klasse, der indeholder alle de aspekter de forskellige kontotyper har til fælles. Den nedarves fra i klasserne "CurrentAccount" og "SavingsAccount". Disse klasser repræsenterer de forskellige typer konti, og indeholder respektive metoder til at håndtere forskellene mellem disse. Ved at benytte denne nedarvningsstruktur kan de enkelte "Account"-underklasser holdes ganske kortfattede.

Tilbage er "Transaktion"-klassen der binder to konti sammen, sikrer validiteten af beløb og typer af konti.

Bilagsliste

Bilag 1 – Burndown Chart, Sprint 2.

Bilag 2 – GUI Labyrint.

Bilag 3 – E/R Diagram

Bilag 4 – Class Diagram