Eksamensopgave

Studienummer og navn på den studerende:

S987155 Mette Willemoes Schrøder Berg

Navn på modul:

62T01 Grundlæggende Objektorienteret Programmering F24

Navn på underviser:

Henrik Tange

Antal anslag inkl. mellemrum:

11 normalsider á ca 24000 anslag inkl. mellemrum

Dato for aflevering:

20. maj 2024

Mette miniBank

Eksamensopgave

62T01 Grundlæggende Objektorienteret Programmering F24

Af

Mette Berg S987155

DTU IT Diplomuddannelsen

Obligatorisk kursus

Indhold

Bilag:

Kildekode til programmet forefindes særskilt i bilag 1.

1. Indledning

Med over 20 års erfaring i finansbranchen er jeg vant til at se bankmedarbejder arbejde med de transaktionsmenuer, der findes i ældre Main-Frame-systemer.

Jeg arbejdede tidligere i Danske Bank, hvor disse systemer stadig er i brug. Meget af den bagvedliggende kodebase kører stadig med oprindelige PL/1 kode, der ikke er objektorienteret. Med C#, vil jeg nu genskabe disse enkle konsolmenuer, som er så nemme for rådgiverne at benytte, fordi der ikke benyttes mus, men i stedet indtastning af menupunkter. Dette er utroligt effektivt at bruge ved mange gentagne manuelle operationer, da talkombinationer for valg af menupunkter hurtigt ligger på ryggraden, og det er muligt at lave talkombinationer for hurtigt at komme dybt ned i en menu-strukturer.

2. Problemformulering

En kunde henvender sig til banken og ønsker hjælp af en rådgiver til at udføre de mest anvendte transaktioner inden for bankvirksomhed: Indsætte, overføre og hæve penge fra og til en bankkonto. Der ønskes lavet et program, hvor bankrådgiveren kan udføre transaktionerne på vegne af kunden, i et menu-system betjent uden brug af mus.

3. Afgrænsning

- Kunder og deres konti er allerede oprettet, der kan ikke oprettes kunder eller konti.
- Ingen desktop UI, i.e. ingen programmering med WFP
- Programmet gemmer ikke data hverken i form af fil eller database
- Rådgiveren skal ikke logge ind og der er dermed ingen sikkerhedsforanstaltninger
- Fravalg af renteberegninger, - eg. rentetilskrivning.
- Fravalg af abonnementsgebyrer (f.eks. månedligt for EUR konto, og erhvervskonto.
- Valutaveksling, dvs. når der overføres mellem EUR og DKK konti vælges en fast vekslingsrate.
- Det er ikke muligt at låne penge af banken.
- Der er to typer af valuta: DKK og Euro.
- En konto kan kun ejes af én kunde.

4. Analyse

Programmet udvikles som et Proof-of-Concept (POC). Med en POC vil jeg kunne se hvad virker og ikke virker, og dermed have lejlighed til at ændre på grundstruktur (ude af defineret omfang for denne opgave) og så stille og roligt udbygge programmet, og addere flere og flere funktioner. men skal kunne skaleres og udvides med flere funktioner.

Der modelleres med to typer af kunder, - privat person og virksomhed eller organisation. Sidstnævnte slås i første omgang sammen til at være én type. En kunde vil have karakteristika som kunde-id, navn, adresse, og et unikt nummer (CVR eller cpr, der benyttes i fm. med offentlig verifikation), og hvorvidt det er en person eller en virksomhed. Typen af kunde er i øvrigt ekstremt vigtigt for KYC processen i en bank, så det forventes at der vil komme mange datapunkter omhandlende dette på senere stadie. Herudover dato for oprettelse af kunden og nedlukning af kunden.

Jeg vil starte med at modellere i første omgang to typer af bankkonti, - en opsparingskonto og en forbrugskonto, som skal markeres for hvilke typer kunder, den kan anvendes på. For en konto vil der være ud over et unikt konto-id også være balancen / saldo på kontoen, beløbsgrænser, kontotype valutatype og kontoindehaver, som vil være kunde id, samt rentesats. Desuden oprettelse og nedlukning af konto.

Som det sidste område er transaktioner. En transaktionspost skal indeholde unikt transaktions-id, gebyr for transaktion, beløb, valuta, tidsstempel, fra-konto/manuel indsættelse, til-konto/manuel hævning og "net" (beløb minus gebyr).

Det bemærkes at en transaktion vil medføre to aktiviteter.

1. Opdatering af saldo på kontoen, med baggrund i beløbet angivet i transaktionen.
2. Logning af transaktionen, - der skal oprettes et transaktionsobjekt for hver bevægelse der foretager ændringer på saldoen på kontoen.

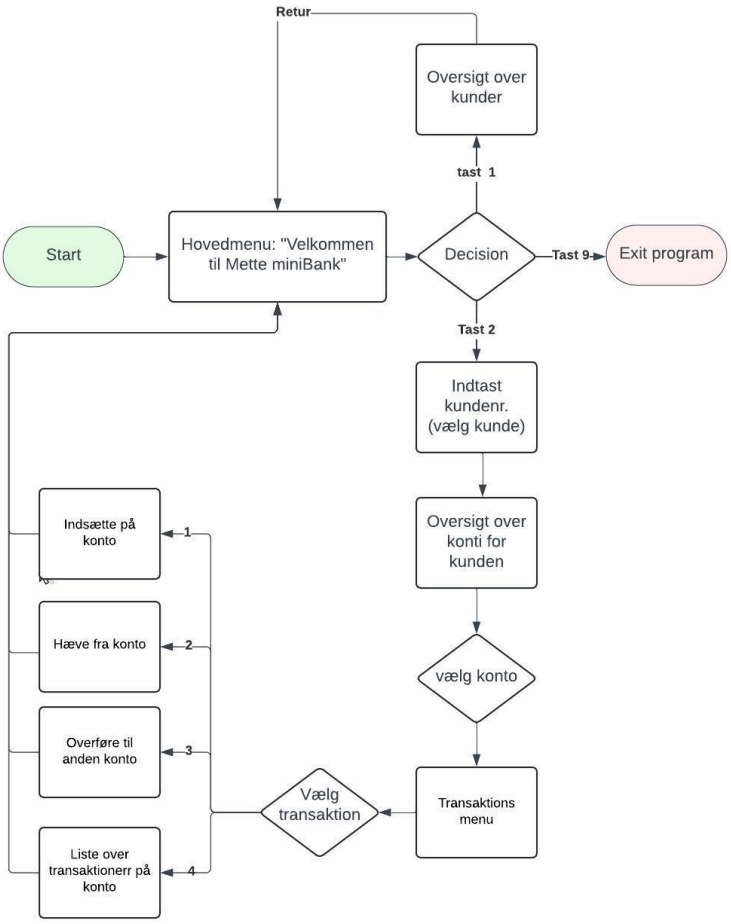
Opdatering af en saldo kan i princippet udføres på kontoen uden at oprette et transaktionsobjekt, men af revisionsårsager, skal der være en meget stærk forbindelse mellem en ændring på en saldo på en konto, og en logning af ændringen. Desuden ville kunden jo heller ikke kunne få en kontooversigt over bevægelser, der er sket på kontoen, hvis transaktionsobjekter ikke blev oprettet.

Særligt i en bank er der strenge sikkerhedsmæssige krav til dataopbevaring, samt krav om GDPR overholdelse. Det er vigtigt fra start af at tænke dette aspekt med ind i programmet, da det har stor betydning for den programopbygning i moduler.

5. Design

5.1 Brugergrænseflade

Da det skal være et menu-system betjent uden brug af mus er løsningen at give kunderådgiveren mulighed for via en konsolapplikation at udføre transaktioner: Indsætte og hæve et kontantbeløb overføre mellem to konti, samt en oversigt over transaktioner på valgte konto, som det fremgår af figur 1 Menu-struktur



Figur 1 Menu struktur for programmet Mette miniBank

5.2 Løsningsforslag til kildekode

Programmet opbygges til at kunne blive skaleres med mange forskellige typer kunder og typer af konti, og derfor er konto og kunde klasserne defineret med en baseklasse med tværgående egenskaber, og med **nedarving** til subklasserne for hhv. konti-typer og kundetyper, hvor specifikke egenskaber for typen så defineres.

Abstraktion er dermed er anvendt for kunder og konti: Baseklasserne fungerer som konceptuelt grundlag for subklasserne, og har i øvrigt fået adgangs-modifikatoren ”Abstract”, idet det ikke skal være muligt at instantiere baseklasserne.

På tværs af kontotype, skal der indsættes og hæves. Men, måden hvorpå dette sker – beregningen – kan variere alt efter kontotype, f.eks. skal der påføres et gebyr, eller er der tale om en valutaveksling. Derfor oprettes et interface for transaktionstyper.
Det bemærkes, at en overførsel svarer til først at udføre transaktionen ”hæve” og derefter ”indsætte”.

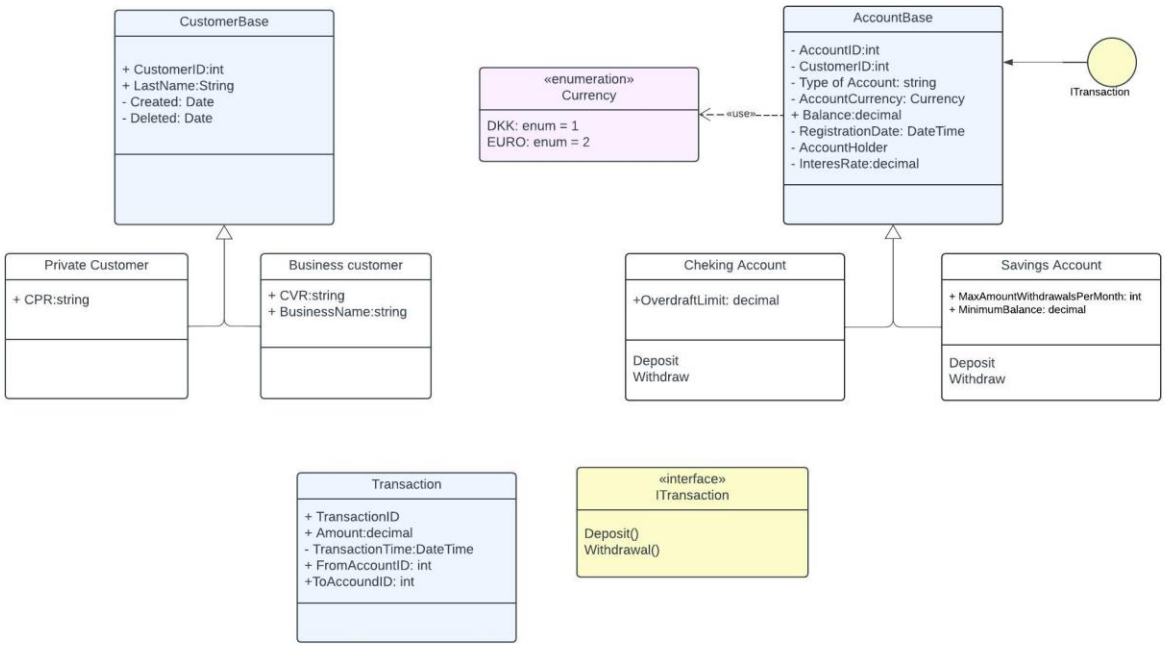
Ved brug af et interface er benyttet **polymorfisme** til transaktionsmulighederne ”indsætte”, ”hæve” og ”overføre”. Interface er brugt, idet indsætte, hæve og overføre er almengyldigt for alle konti (hvor programmet fremadrettet skal rumme ikke blot forbrugskonto (eng.:”Checking account”), men også opsparingskonto (eng: ”Savings account”), investeringskonto, børneopsparingskonto osv. Disse

konti har individuelt forskellige beregning af gebyr og rente ved en transaktion. Ved at benytte interface kan transaktionsmulighederne benyttes for alle typer subklasser af konti (og dermed behandles ens), men kan blive implementeret forskelligt på de forskellige konti type, der vil have forskellige beregninger for gebyr, rente og abonnement.

Dette tillader udskiftning af implementeringer (eg. en specifik beregning af en rente på en toprentekonto) på et senere tidspunkt uden at ændre resten af kodebasen, hvilket gør koden fleksibel og genbrugelig. Dermed

Indkapsling skal i programmet anvendes systematisk så dele af data i koden skjules, og ikke kan opsnappes. Indkapsling defineres som indpakning af data i en enkelt enhed, samt styring vha. adgangs-modifikationer af hvordan andre enheder i kodebasen, der har adgang til data. Dette vil skabe tydeligt definerede grænseflader, hvormed datafølsomme oplysninger vil blive beskyttet. **Indkapsling** skal derfor iagttages i høj udstrækning i programmet ved at lægge kode i moduler, og ved at bruge adgangsmodifikatorer. Som en absolut god sideeffekt bliver koden også mere organiseret da den bliver modulær og dermed mere vedligeholdelsesvenlig.

Klasser og interface designes derfor som skitseret på figur 2, hvor sammenhænge også er optegnet:



Figur 2 Skitse over klasserne og afhængigheder for programmet Mette miniBank

Det engelske sprog benyttes i selve kildekoden, for at sikre, at det er læsbart af flere nationaliteter. Der benyttes derfor udelukkende engelsk til variabler og konstanter, - ord som ”saldo på konto” oversættes til ”balance” og beløb på transaktion oversættes til ”amount”. Kommentarer skal være på engelsk. En forbrugskonto oversættes til ”chekings account” og en opsparingskonto til ”savings account”. Transaktionsbeløb benævnes som ”amount”, og kontosaldo/kontobalance ved den engelske betegnelse ”Balance”.

Programmet kodes i C# med brug af de fire grundlæggende principper for objektorienteret programmering: Abstraktion, polymorfisme, indkapsling og [1][2], og søges understøttet af Retningslinjer for Clean Code [6][7][8] og SOLID principperne [3][4] [10]. Dette er nærmere undersøgt og beskrevet i teoriafsnittet kapitel 8.

6. Implementering

6.1 Opbygning

Følgende discipliner er benyttet i programmet

- En kollektion af den generiske type Lists<T> (hvor typen T varierer alt efter klasse anvendt på objekter, eg. type ”Transaction” eller type ”konto”)
- Bruger input og validering i menu strukturen.
- Flow Kontrol (”Switch” er benyttet til menu systemet)
- Interface til at håndtere transaktioner og aktivitet på alle konti.
- Valuta defineres som enumeration liste af konstanter.

Programmet er kodet i C# og IDE Visual Studio. Alle klasser er placeret i egen fil .cs med eget specifikke ansvar, og det er kun menuer og styring af disse, der ligger i Main. Klasser og logik er grupperet og lagt ind i dertil navngivne typer af mapper/namespaces. Mappeopbygning og namespaces følges ad i dette program. Dette er ikke en nødvendighed, men det er god stil i forhold til læsbarhed, da det også er med til at tydeliggøre struktur af programmet. Med modulopbygningen opnår jeg overblik, struktur og adskillelse af ansvar og mulighed for genbrug, også i andre programmer på sigt.

Selve opbygningen af programmet er beskrevet i nedenstående tabel:

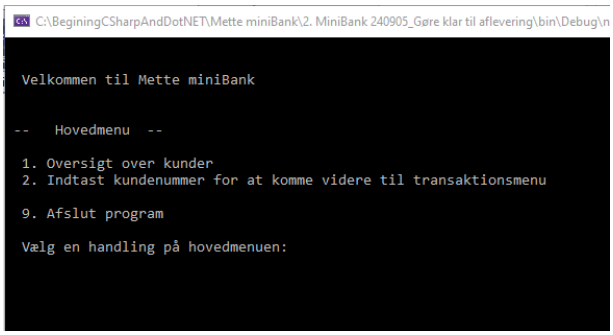
Mappe	Indhold	Formål, indhold
/..	Mette_miniBank.csproj	Samler projektet. Udgangspunkt for Visual Studio.
/..	Program.cs	**namespace Mette_miniBank** Hovedprogram. Indeholder menu-struktur for brugergrænsefladen – en consol.
/../Engines/	TransactionEngine.cs	**namespace Mette_miniBank.Engines** Indeholder forretningslogik. Her ligger transaktionslogikken.
/../Interfaces/	ITransaction.cs	**namespace Mette_miniBank.Interfaces** Her ligger alle interfacene. Indtil videre kun én, ITransaction, med tre metoder - hæve, indsætte og indhente transaktionspost id.
/../Types/	BankAccountBase.cs BusinessCustomer.cs CheckingAccount.cs CurrencyEnum.cs CustomerBase.cs PrivateCustomer.cs Transaction.cs	**namespace Mette_miniBank.Types** Her ligger alle klasserne, dels hovedklassen for kunde og konto, og dels subklasserne for de to typer af kunder og de to typer af konti. Herudover valutaliste og klassen for transaktioner.
/../ Repositories	AccountRepository.cs CustomerRepository.cs TransactionRepository.cs	**namespace Mette_miniBank.Repositories** Her oprettet konti og kunder og lister for disse defineres. Endvidere metoder og lister for transaktioner.

Figur 3 Opbygning og filstruktur af programmet

I appendiks kan opbygningen/strukturering ses et skærmdump fra Visual Studio samt mappestruktur i Explorer.

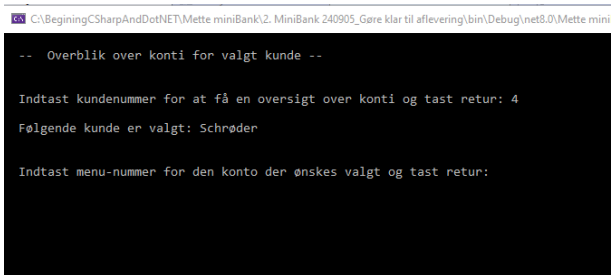
Programmet opretter tre private kunder, med et par konti hver. Konti og kunder ”lever” kun så længe programmet lever, idet det ikke gemmes/hentes fra en database eller en fil i sin nuværende version.

Programmet starter en konsol med en hovedmenu med to muligheder: Oversigt over kunder i banken, og mulighed for at udføre en transaktion på vegne af kunden.

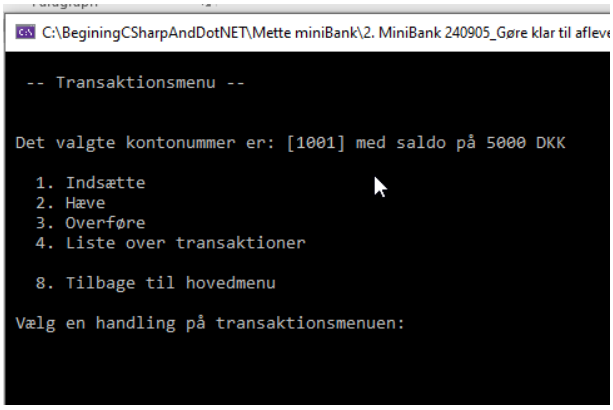


Figur 4 Hovedmenu

Menuerne er bygget hierarkisk op, således at først skal vælges kunde, derefter kunde og så skal vælges hvilken transaktion der skal forgå (for overblik se figur 2 under ”Design”):



Figur 5 Menu "Overblik over konti for valgt kunde"



Figur 6 Transaktionsmenu

For yderligere at se hvorledes transaktionsmenuen fungerer henvises til appendiks: ”Transaktionsmenu”.

6.2 Bemærkninger til afleveret kildekode

Jeg har følgende bemærkninger til ændringer, for at programmet kan fremstå afsluttet i en helhed og dermed kan bedømmes som et Proof-of-Concept produkt:

Ikke implementeret: Der er kun implementeret forbrugskonto, ikke opsparingskonto. Virksomhedskunder er ikke implementeret (klassen er oprettet). Adgangsmodifikatorer er ikke gennemført systematisk.

Læsbarhed af koden: Kommentarer er på dansk og burde være på engelsk, for at sikre kommende IT udviklere kan benytte kommentarer til at forstå programmet.

Kendte fejl i programmet: Når der ingen bankkonti er tilknyttet en kunde, og man vil overføre konti til denne, kommer man ind i et uendeligt loop, man ikke kan komme ud af. Det skal der udvikles kode til. Det er desuden muligt at overføre forskellige beløb mellem konto uden at tjekke for at valutatypen er ens for begge konti.

Nuværende begrænsninger: Der kan kun overføres internt mellem en kundes egne konti og ikke mellem konti eget af forskellige kunder.

Overvejelser omkring menu-struktur: Den nuværende menu-struktur er ikke optimal. Den komplicerer koden, fordi den er hierarkisk opbygget, selvom der er genbrug af elementer. Dette har for eksempel bevirket, at overførsel mellem konti kun sker pt. mellem konti ejet af samme kunde. Heldigvis er ansvar skarpt adskilt, og opbygning af en ny menu struktur påvirker ikke klasserne, forretningslogik og repositories, der jo ligger for sig selv. Dermed kan alle elementer genbruges i en ny og fladere menu-struktur.

Overvejelser omkring brugervenlighed
Der er for meget skift mellem hvornår der skal indtastes et tal for et menupunkt med og uden <retur>, hvornår det er et kunde-id-nummer, der skal indtastes. Ved fejlindtastning skaber dette forvirring, fordi der så skal indtastes retur for at komme tilbage. Denne opsætning skal gentænkes igen, så der fremstår mere ensartethed.

7. Test af program

Unit test: Der er foretaget løbende test af moduler i programmet, og fejl i koden er til rettet løbende.

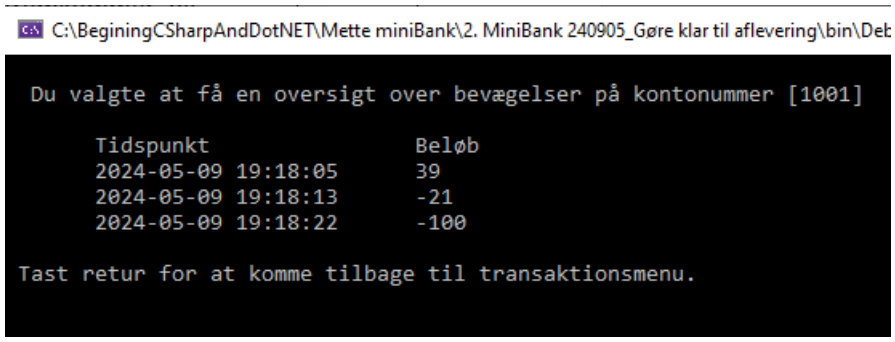
Integrationstest: Da programmet ikke interagerer med andre programmer eller en database, er der ikke foretaget integrationstest.

Systemtest: Overordnet set opfylder programmet de funktionelle krav, der er stillet indirekte under kapitel 2: Problemformulering. Der er ikke stillet nogen ikke-funktionelle krav til programmet. Performance på programmet er dog testet subjektivt, og opleves acceptabelt, da der ikke opleves nogen bemærkelsesværdig reaktionstid, når der interageres med programmet ved indtastning af valg.

Brugertest: Der er udført en række brugertest, hvoraf tre nævnes i nedennævnte tabel:

Brugertests – id på testcase og navn	Beskrivelse af test	Ønsket output	Resultat af test
Testcase 1.1: Menu ”overblik over konti” - Test af indtastning af kunde ID	Hvilken besked får kunden, når der indtastes et heltal, som ikke eksisterer i kundebasen?	Sætningen "Der findes ingen kunder på det indtastede nummer." fremkommer, og kunden ledes tilbage til menu, ved at taste på en tilfældig tast.	ok
Testcase 1.2: Menu ”overblik over konti” - Test af indtastning af kunde ID	Hvilken besked får kunden når der indtastes et bogstav og ikke et heltal?	Sætningen " Det indtastede kundenummer var ikke korrekt - det skal være et heltal." fremkommer, og kunden ledes tilbage til menu, ved at taste på en tilfældig tast.	ok
Testcase 3: Oversigt over bevægelser på konti – Test af data i transaktionslisten	For kunde id 2, konto 1001 indsættes 39 kr, overføres til konto 1002 100kr og hæves 21kr. Test at	Listen viser: +39 kr. -21 kr. -100 kr.	Ok – se vedlagde skærmdump

Figur 7 Eksempel på testcases udført



:

Figur 8 Resultat af testcase 3

8. Teoriafsnit: ”Clean Code”, SOLID og de fire søjler for OOP

Jeg talte for nylig med en C# programmør i Danske Bank, der tidligere har siddet med PL/1 kode. Han fortalte, at da man begyndte at ”outsource” funktioner til Indien, mødtes de af det problem, at al PL/1 koden – alle variabler, konstanter og kommentarer, var beskrevet på dansk. Indiske IT udviklere havde ikke en chance for at kunne læse koden, og dermed blev banken begrænset i dens ønske om ressourceoptimering og hente arbejdskraft udefra.

Det er vigtigt at etablere og benytte god programmeringspraksis. Ikke blot at lave funktionel og performanceoptimeret kode, hvilket sælger sig selv, men også at kunne lave læsbar kode, der er nem at forstå for andre programmører, at lave kode der er er nem at vedligeholde og ændre i. Som med alt andet er det nemt at komplicere tingene – koden, men faktisk svært at simplificere og gøre enkelt. Det er omkostningsoptimering på sigt.

Programmering er en sjov og kreativ proces, og man bliver hurtigt opslugt i arbejdet, og det er yderst tilfredsstillende med de synlige resultater af ens kreative arbejde. Så det kan føles omkostningstungt og opremsende at skulle holde sig selv tilbage ved at bruge tid på strukturering og god praksis, - og i et team at overholde de kodningskonventioner, der er besluttet [6].

Den selvdisciplin der ligger i dette, ligger ikke altid lige nemt for, når man programmerer. Det er er vigtigt med bevidsthed omkring, hvad konsekvenser det har på et senere tidspunkt, hvis man *ikke* gøre det. Heldigvis er der hjælp at hente i regler og principper, nemlig ”De fire søjler”, ”Clean Code” samt ”SOLID”.

For at et sprog kan være objektorienteret, skal det bygges op med klasser, metoder, objekter og attributter og er bygget på fire grundlæggende principper ””De fire søjler”) for objektorienteret programmering: Abstraktion, polymorfisme, indkapsling og nedarving (eng. ”Abstraction”, ”polymorphism”, ”encapsulation”, ”inheritance”). [1][2][9]

”SOLID” (Fem principper for objektorienteret design OOD) og ”Clean Code” (en række retningslinjer for kode ved objektorienteret programmering OOP) er to tæt sammenbundne begreber, der kan hjælpe os med at skrive OOP kode, der er læsbar, forståelig, robust og nem at vedligeholde og udvide for andre IT udviklere. ”Clean Code og SOLID principperne er i øvrigt introduceret (udviklet og samlet sammen fra ideer fra andre IT udviklere) af Robert Martin [3][4][5][6].

Her har jeg valgt at opsummere fem vigtige ”Clean Code”-retningslinjer for OOP [6][7][8]:

Enkelt ansvars princippet: Hver klasse skal have et enkelt ansvar. Dette betyder, at klassen skal fokusere på én ting og gøre det godt. Hvis en klasse har flere ansvarsområder, skal den opdeles i mindre klasser.

Ingen duplikering: Undgå at duplikere kode. Hvis du har den samme kode flere steder, skal du finde en måde at abstrahere den ud til én enkelt placering

Simplicitet: Denne regel er rigtig svær. Det kræver analyse og ydmyghed at skrive simpel og letforståelig kode. Det er nemt komplekse konstruktioner og unødvendig kompleksitet ved ikke at give sig tid til at tænke sig om.

Høj læsbarhed: Skriv kode, der er let at læse og forstå for andre programmører. Der skal bruges beskrivende navne, god indrykning og kommentarer, selvom kommentarer i princippet kan udelades, hvis koden er læsbar. Det mest oplagt er selvfølgelig at bruge meningsfulde navne for variabler, funktioner, klasser og metoder (gerne følge de normale konventioner med hvornår der bruges PascalCase, camelCase, ALL_CAPS).

Omstrukturering/refaktorering: I ens egen iver efter at få lavet funktionalitet og se det virke, kan det være en overvindelse at have fokus på ”Clean Code” også. Det er derfor god skik at inkorporere de gode vaner løbende, - eller som minimum med regelmæssige intervaller standse op og omstrukturere – for at sikre at koden ændres for at forbedre dens struktur, læselighed eller vedligeholdelse uden at ændre dens ydre adfærd. Ting har det med at hobe sig op, og det nødvendigt med løbende oprydning både derhjemme i ens hus, men også i kode.

Test: Skrive test til koden for at sikre, at den fungerer korrekt. At skrive test fører til, at man tænker lidt mere over design af koden, fordi man opdager, hvorfor den kan være svær at test. Med test kan man finde fejl og defekter, som man ikke opdager under manuel testning. Og som en vigtig del, det giver mere tillid til at ændre koden, uden at funktionalitet brydes. Derved bliver det nemmere og udvikle på koden.

"Clean Code"-retningslinjerne og SOLID-principperne kan gå hånd i hånd for at skabe struktureret, letforståelig og fleksibel kode. SOLID principperne nævnes her i Figur 9 og på engelsk:

		Navn på princippet	Essensen af princippet
S	SRP	Single Responsibility Principle	Split it (don't overdo it either)
O	OCP	Open/Closed Principle	Extend it (preferably abstraction)
L	LSP	Liskov Substitution Principle	Don't fake it (preferably compose it)
I	ISP	Interface Segregation Principle	Don't enforce it (compose or inherit)
D	DIP	Dependency Inversion Principle	Inject it (don't hard-core dependencies, use abstractions)

Figur 9 S.O.L.I.D principperne for objekt orienteret design

SOLID-principperne for design spiller en central rolle, når det kommer til overholdelse af de grundlæggende principper (de fire søjler [1][2]) for objektorienteret programmering, som har udviklet sig siden starten af 1960'erne [9]. Samspillet er på følgende vis:

S. Enkeltansvarsprincippet (SRP): SRP opfordrer til **indkapsling** ved at fremme klasser med en enkelt opgave, hvilket hjælper med at håndtere kompleksitet og opretholde en klar adskillelse af interesser inden for kodebasen.

O. Åben/Lukket-princippet (OCP): Følger **nedarvning** og **polymorfi** ved at opfordre til, at klasser er åbne for udvidelse, men lukkede for ændring. Ved at bruge nedarvning og polymorfi kan ny funktionalitet tilføjes til systemet uden at ændre eksisterende kode, hvilket fremmer genbrug og udvidelighed af kode.

L. Liskovs substitutions-princippet (LSP): Følger **nedarvning** og **polymorfi** ved at sikre, at objekter af en superklasse kan erstattes med objekter af dens underklasser uden at påvirke programmets korrekthed. Dette fremmer polymorfi ved at tillade, at objekter behandles ens gennem deres fælles basistype og samtidig respekterer adfærden defineret af deres specifikke typer.

I. Interface Segregation Princippet (ISP): Følger **abstraktion** og **polymorfi** ved at fremme brugen af klient-specifikke grænseflader skræddersyet til klienternes behov, hvilket hjælper med at opnå abstraktion ved at skjule unødvendige detaljer. Ved at definere snævre og fokuserede grænseflader letter ISP polymorfi ved at tillade, at objekter interagerer gennem fælles grænseflader uden at afhænge af unødvendige metoder eller adfærd.

D. Afhængighedsinversionsprincippet (DIP): Følger **abstraktion** ved at fremme løs kobling mellem klasser ved at afhænge af abstraktioner i stedet for konkrete implementeringer. Ved at stole på grænseflader og abstrakte klasser hjælper DIP med at opnå abstraktion ved at skjule implementeringsdetaljer og fremme fleksibilitet i kodebasen.

Reglerne for objektorienteret programmering, når de følges med god kodepraksis og SOLID-principperne, fremmer en robust og vedligeholdelsesvenlig kodebase. Ved at fokusere på disse retningslinjer og principper, opnår IT udviklere kode, der er lettere at forstå, vedligeholde og udvide.

Disse principper fungerer som en solid struktur, der hjælper med at skabe fleksible, modulære og robuste softwareløsninger.

Når man følger retningslinjerne for kodeprincipper og SOLID-principperne i objektorienteret programmering, får man en mere overskuelig, robust og vedligeholdelsesvenlig kode. At følge principperne og retningslinjerne for "Clean Code" hjælper med at organisere koden, så den er lettere at forstå, ændre og udvide. Det er som at bygge med LEGO-klodser – man har klare regler for, hvordan klodserne skal sættes sammen, så man kan lave store og komplekse strukturer, samtidig med at man holder det hele overskueligt og fleksibelt.

Dermed er det også muligt i en virksomhed at kunne optimere på IT ressourcer, - F.eks. at kunne outsource udvikling af koden til en anden virksomhed.

9. Konklusion

Jeg har lavet en såkaldt POC – en Proof-of-concept - til et transaktionsprogram til en bank, "Mette miniBank", hvor der kan overføres beløb mellem konti, indsættes kontanter og hæves kontanter af en bankrådgiver. Programmet kan eksekvere på de mest basale principper for bankoverførsel. Principper for objektorienteret programmering er benyttet, og opbygningen tillader udvidelse med flere kundetyper, konto-typer samt funktioner.

For at sikre at kildekoden kan læses af andre eller IT udvikleren selv, for at kunne fortsat udvikle på koden, eller vedligeholde, - eller benytte koden i andre programmer, er der begrundet vigtigheden af vedtage og overholde kodningskonventioner. Principperne bag "Clean Code" og "SOLID" er taget med i overvejelserne ved udvikling af kildekoden.

Programmet, jeg har udviklet, danner en god basis for at fortsætte min egen uddannelse inden for programmering og benytte det som udgangspunkt til at lære mange forskellige discipliner, f.eks. lave et UI interface i form af en netbank eller en rådgiverplatform ved at bruge Windows Presentation Foundation (WPF) modulet af .net, arbejde med services i form af WCF eller arbejde med Web API så data fra f.eks. transaktioner kan listes i en anden applikation. Og ikke mindst lave et sikkert login for både bruger og rådgiveradgang samt generel IT sikkerhed.

Herudover skal der etableres styringsmekanismer for metadata – f.eks. laves versioneringskontrol vha. GitHub.

Det næste naturlige skridt er dog at udvide programmet med kunne at gemme og hente data i og fra en SQL database, at rådgiveren kan oprette og slette kunder samt konti, og at kunne overføre beløb mellem konti mellem forskellige kunder, og ikke mindst skrive en saldooversigt ned på fil.

Litteraturhenvisning

Om de fire søjler i objekt-orienteret programmering (OOP):

1. Microsoft Lean:
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop

2. Hjemmeside Free code camp:
https://www.freecodecamp.org/news/four-pillars-of-object-oriented-programming/

Om SOLID principperne:

10. Hjemmeside:
https://www.quora.com/What-are-SOLID-design-principles-1

3. Essay: *Martin, Robert C. (2000). "Design Principles and Design Patterns" (PDF). www. objectmentor.com. Archived from the original on 2015-09-06.*

4. Bog: *Martin, Robert C. (2002)*: Agile Software Development Principles Patterns and Practices. SOLID principperne forklares fra si 95

5. Hjemmeside Free code camp:
https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

Om "Clean Code":

6. Bog: Robert C. Martin: "Clean Code*: A Handbook of Agile Software Craftsmanship*"

7. Sammendrag af bogen Clean Code af Robert Martin:
https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

8. Webside "Codegym" om kodningskonventioner og hvordan man får gode kodeteknikker (ganske vist med udgangspunkt i JAVA, men principperne er jo det samme)

https://codegym.cc/da/groups/posts/da.387.10-mader-at-forbedre-din-kode-pa-bevist-gennem-personlig-erfaring

Historien om objektorienteret kode og hvordan de fire søjler er blevet introduceret

9. Hjemmeside: Historien om objektorienteret programmering
https://en.wikipedia.org/wiki/Object-oriented_programming

Appendix

Strukturering af programmet – skræmdumps fra Visual Studio og Explorer

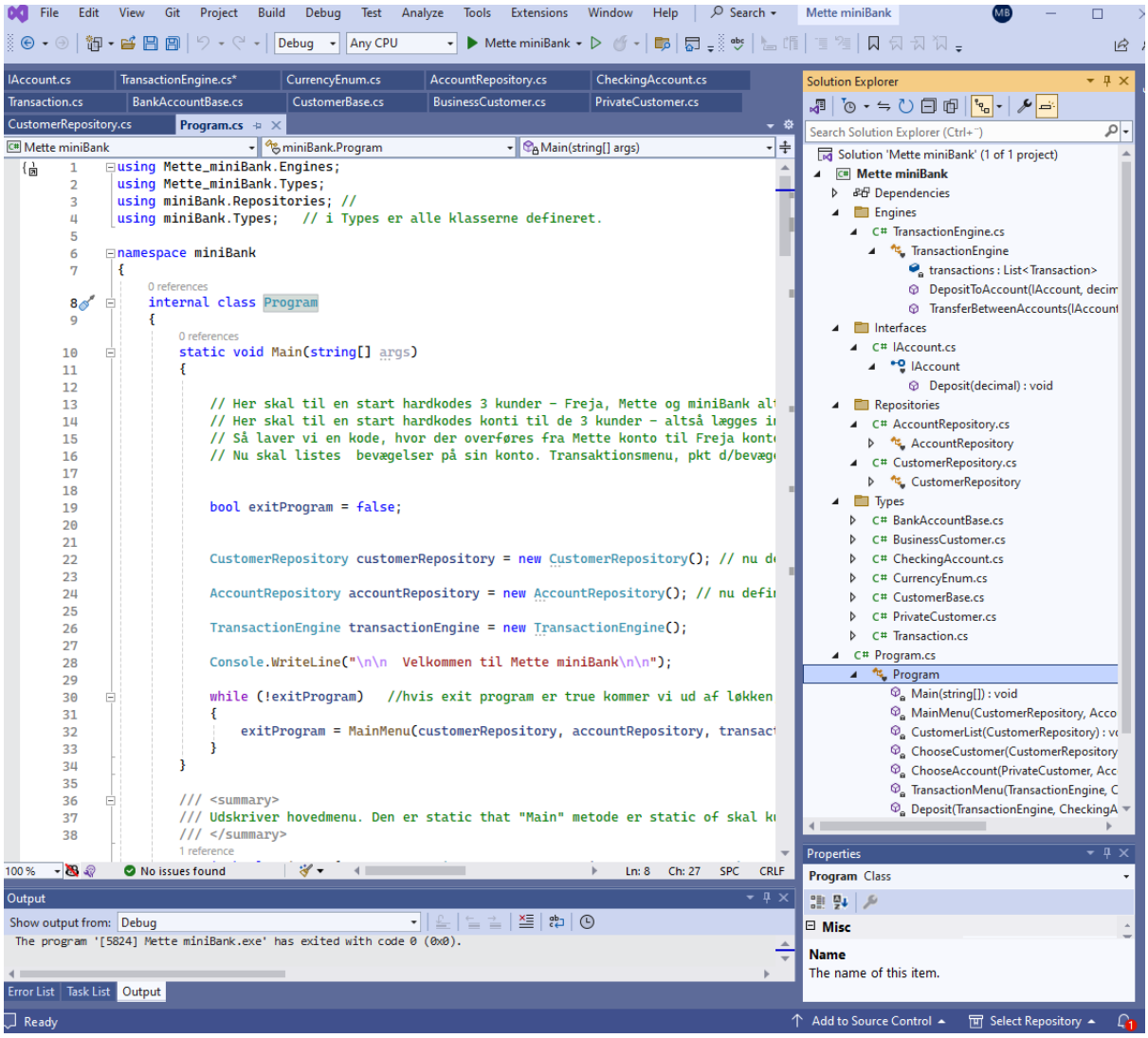


Figure 10 Programmets opbygning i Visual Studio

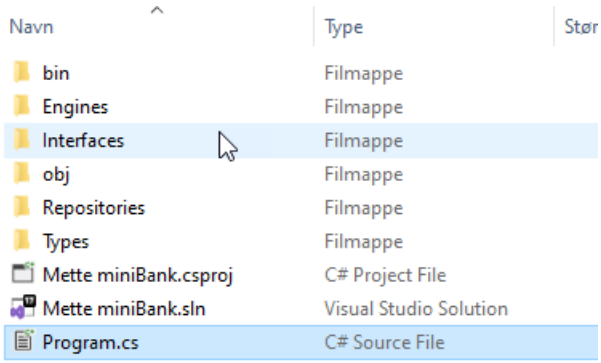
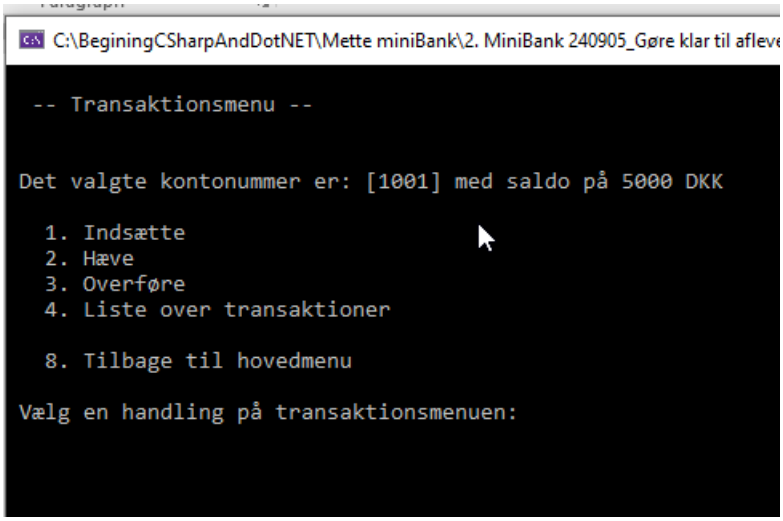
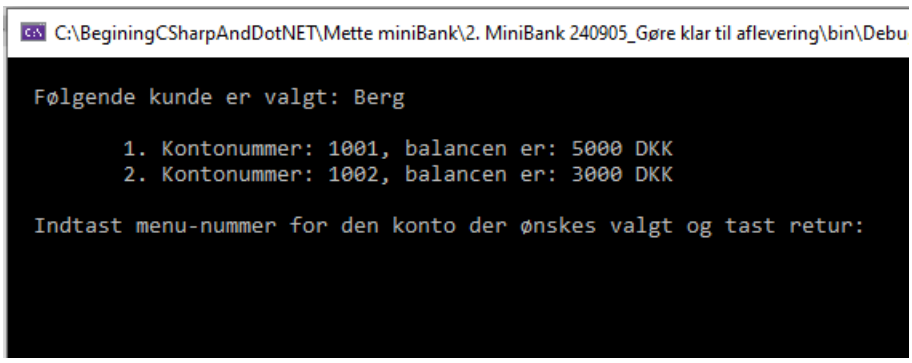


Figure 11 Programmets opbygning i Explorer

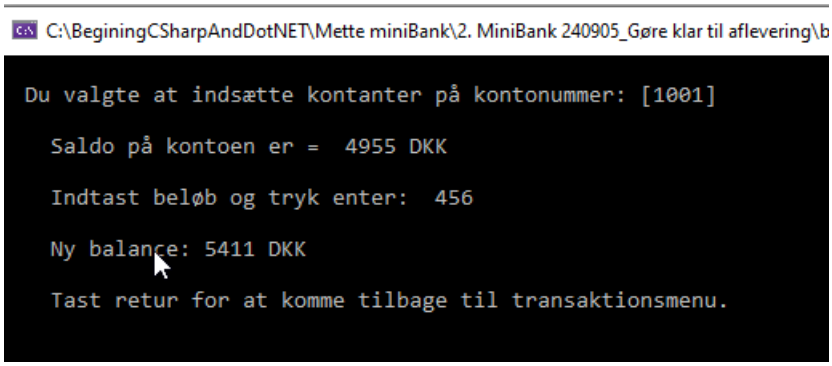
Transaktionsmenuen – eksempel på transaktioner i menuen





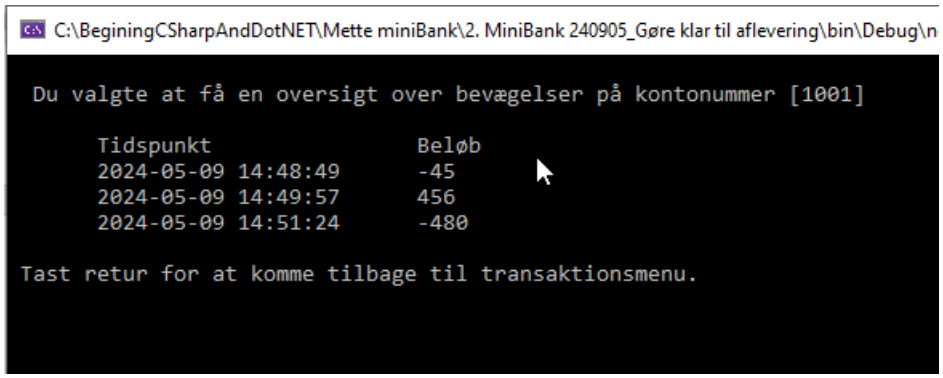
Overføre:

Indsætte kontanter:

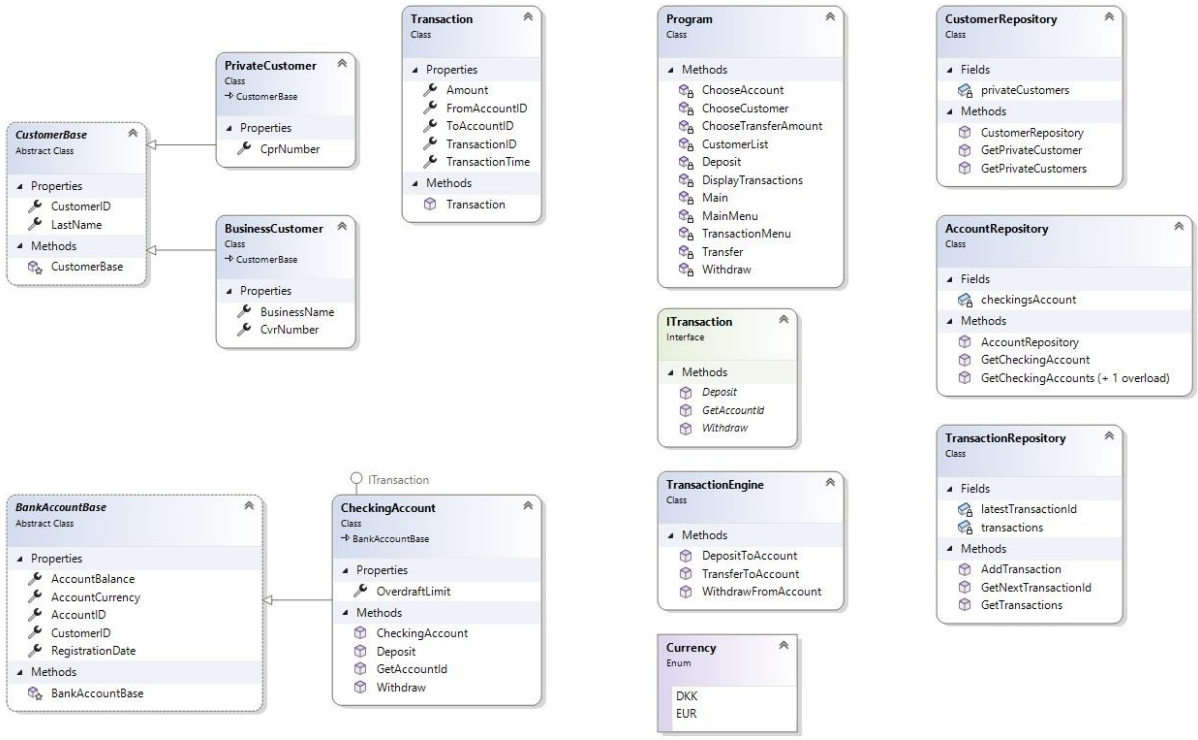


Hæve kontanter:

Oversigt over transaktioner:



Klasse diagram fra seneste version af det implementerede program



Figur 12 Klasse diagrammet skabt i Visual Studio