

# Projektbeskrivning

## Budgetprogram

2032-03-22

### Projektmedlemmar:

Carl Eklund [carek123@student.liu.se](mailto:carek123@student.liu.se)

### Handledare:

Handledare erima882@student.liu.se

## Innehåll

1. Introduktion till projektet[OBJ]2
2. Ytterligare bakgrundsinformation[OBJ]2
3. Milstolpar[OBJ]2
4. Övriga implementationsförberedelser[OBJ]4
5. Utveckling och samarbete[OBJ]4
6. Implementationsbeskrivning[OBJ]6
  - 6.1. Milstolpar[OBJ]6
  - 6.2. Dokumentation för programstruktur, med UML-diagram[OBJ]6
7. Användarmanual[OBJ]7

# Projektplan

## 1. Introduktion till projektet

Mitt projekt kommer att vara ett budgetprogram där man som användare kan se hur långt man har kvar till sin budget samt se hur långt man har kvar på sitt lån. En budget är ett viktigt verktyg i arbete att styra ett företags samt enskilda individers ekonomi. Mitt projekt kommer låta dig som användare att skapa ett konto, därefter lägga in budget, hur mycket du vill spara i månaden, och sedan se hur långt du har kvar tills du når målet. Du kan även som användare lägga in eventuella lån/skulder och därefter se hur långt du har kvar tills lånet eller skulden är betald.

## 2. Ytterligare bakgrundsinformation

### 2.1 MySQL Databas

Mitt projekt började med att jag använde en textfil för att kunna lagra data om de registrerade användarna samt deras lån. Denna textfil användes även vid inloggning för att kunna logga in rätt användare. Eftersom jag har arbetat mycket med databaser och dess koppling till Java innan så gick jag över till att lagra data i en MySQL databas, detta eftersom söktiden samt lagringen blir mycket mindre och smidigare. Men det kan uppstå en liten fördröjning när man vill ladda adminsidan eftersom all information om användarna dyker upp.

### 2.2 MVC Design Pattern

Det används en MVC pattern för att kunna hålla information om den användare som loggar samt även vilket användare som ska få det nyligen tillagda lånet. Detta eftersom jag använder en Cardlayout för att byta mellan de olika sidorna i applikationen. En MVC model består av två komponenter en modell och en view. Där model hanterar att ta in data t.ex. vilken användare som loggar in från logginsidan view:n tar hand om att visa den informationen på en annan sida av applikationen t.ex. huvudsidan där användaren kan se sina lån som kommer efter logginsidan.

Cykeln för ett MVC pattern ser ut som att modellen skickar information till en controller som sedan håller i den informationen tills den ska visas på en annan del av applikationen. Detta underlättar eftersom själva klasser som håller information om hur sidan ska se ut behöver veta vilket användare som är inloggad just nu utan den får reda på det när användare först loggar in.

## 3. Milstolpar

#	Beskrivning
1	En design för varje sida som kommer att vara med i programmet, även ett enklare

	UML-diagram där information om de olika klasser ska stå. En plan över hur alla sidor samt klasser är kopplade
2	Samtliga sidor är implementerade rent designmässigt men saknar funktionalitet och koppling
3	Koppling mellan de olika sidor finns via knappar och liknande
4	En enklare inloggning där det enbart finns en "inloggad" användare åt gången
5	Användaren kan nu skapa lån, samt lägga till i en lista av lån, lånen syns enbart i konsolen
6	Användarens låns syns på de grafiska
7	Användarens inloggningsinformation sparas nu i en textfil, användarens kan nu skapa flera konton vilket sparas i en textfil. Vilket gör att enkel primitiv databas finns
8	Den inloggade användarens lån läses från textfil
9	Användarens inloggningsinformation sparas nu i en databas
10	Användarens låninformation sparas nu i en databas
11	Lägg till en admin sida där man kan ändra användares inloggningsuppgifter
12	En logger klass som används för att logga eventuella fel/information
13	Import/Export funktion
14	Användare kan både ändra sin egen kontoinformation samt ändra sin information om lån
15	
16	
17	
18	
19	
20	

## 4. Övriga implementationsförberedelser

### 4.1 Klasser

Att ta reda på vilka huvudklasser jag behövde samt vilka parametrar som skulle ingå var viktigt eftersom det är dessa klasser som bär applikationen. Eftersom jag valde att budgetprogram så visste jag att jag skulle vilja ha en Användare klass samt en Lån klass.

### 4.2 Övergång mellan sidorna

Sen uppstod problemet att byta sida inom programmet och behålla information från den föregående, detta för att logginsidan samt sidan där man lägger till ett lån ska fungera på ett smidigt sätt. MVC design pattern låter de olika sidorna vara ovetande från vilken användare som används för tillfället utan detta sköter de olika kontrollerna som finns implementerade.

### 4.3 Lagring (Textfil/Databas)

Eftersom jag även ville att programmet skulle kunna lagra flera olika användare samt spara undan användarens lån. Första tanken var att spara användare i en databas men för att kunna komma igång med funktionaliteten snabbare så valdes en textfil som lagringen för att sedan gå över till en databas.

# Projektrapport

## 6. Implementationsbeskrivning

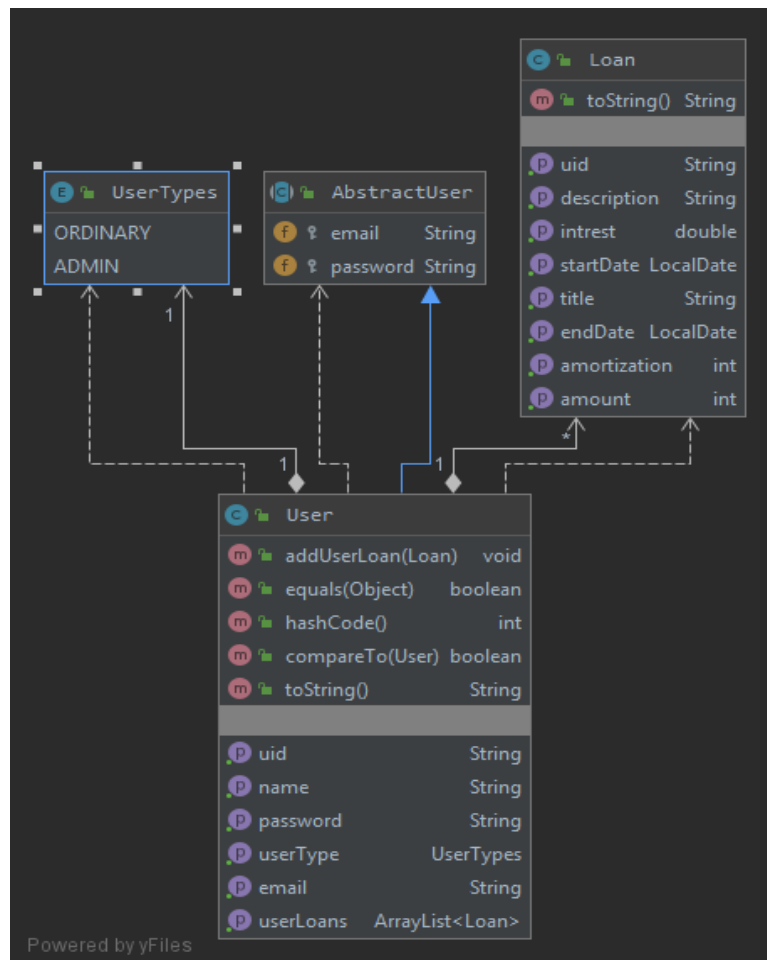
### 6.1. Milstolpar

Från milstolpe 1 till 12 fungerar helt, milstolpe 13 och 14 är inte implementerat än.

### 6.2. Dokumentation för programstruktur, med UML-diagram

#### 6.2.1 Klassrelationer

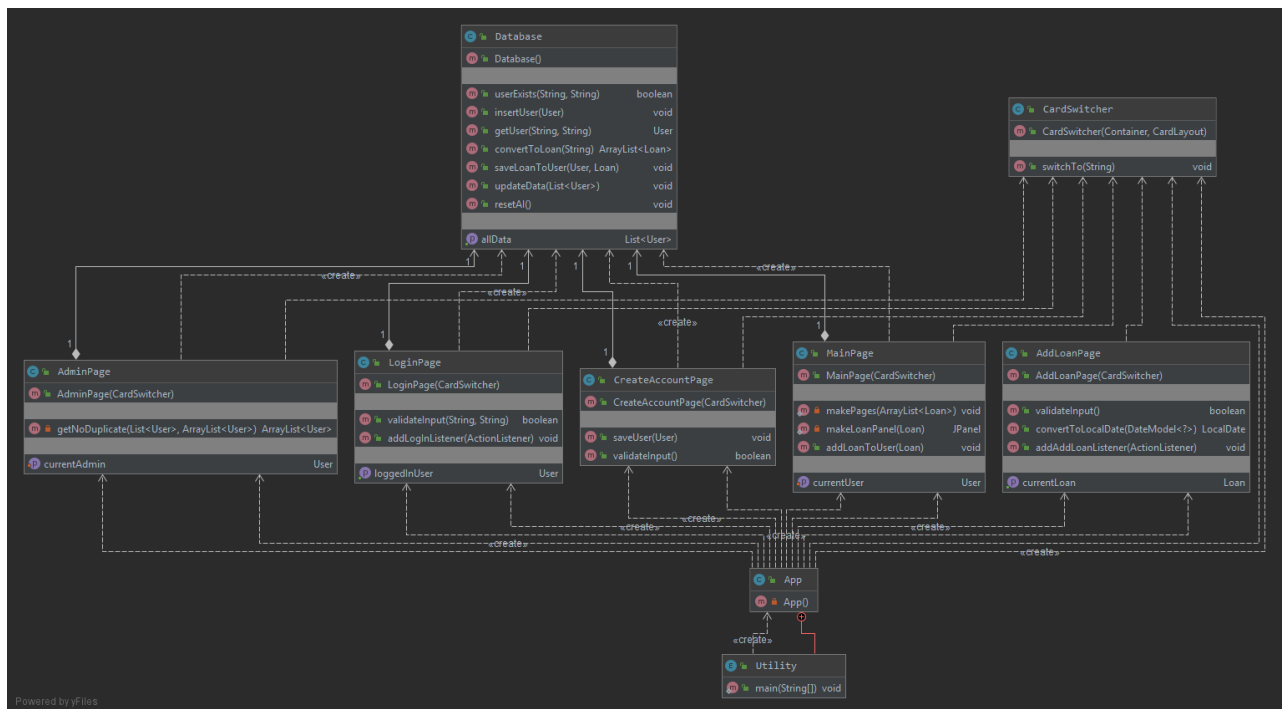
I programmet så finns klassen User som håller inloggningsuppgifter men även en lista av klassen Loan som i sin tur innehåller uppgifter nödvändiga. User klassen ärver även från AbstractUser klassen eftersom varje användare ska ha minst lösenord samt email för att kunna klassas som en User. Det används även Enumtypes för att kunna definiera olika sorters användare vid skapandet av ett konto. Vid utökning så behöver programmaren lägga till den sorters användare som ska användas som ett enumvärde i UserTypes.



UML Diagram för User och Loan

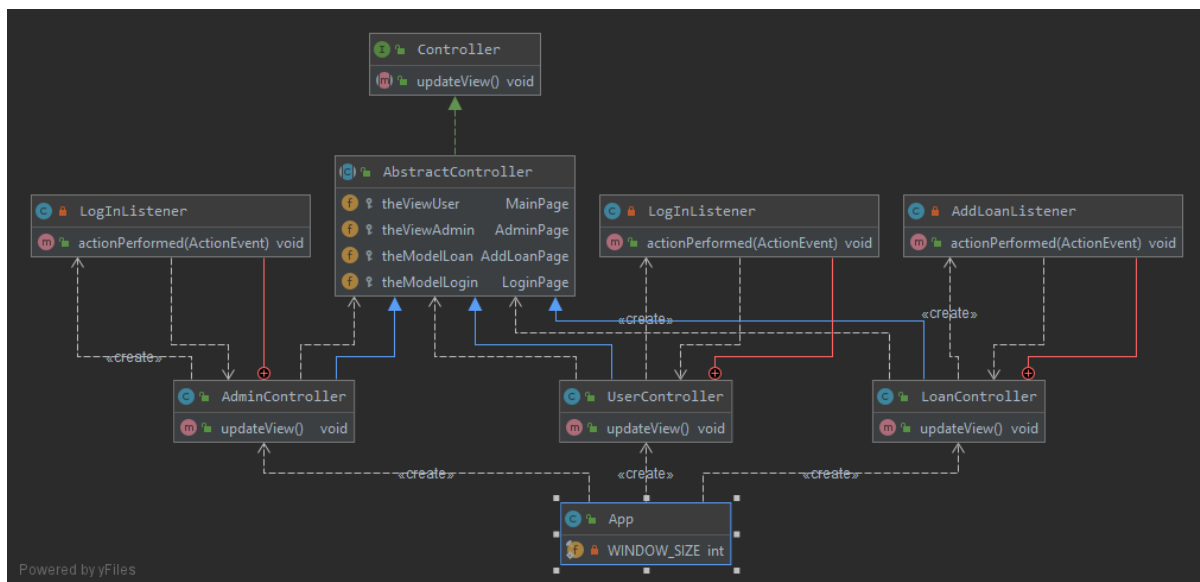
I programmet finns även olika sidor som kan visas för användaren, även en huvudklass som instansierar alla dessa sidor. För att kunna byta mellan de olika sidor användes en Cardlayout där varje sida tolkas som ett kort där huvudprogrammet bara byter kort beroende på vart användare vill gå. Denna funktion har extraherats till en klass, CardSwitcher, som sköter skiftet mellan de olika sidorna denna klass har en switchTo

metod som tar in en layout vilket i detta fall är en Jpanel som instansieras i App klassen (huvudklassen) samt en container vilket är vilken sida som ska visas.



UML Diagram för App, Databas, CardSwitcher samt för de olika sidorna

Vilket visas i UML-diagrammet ovan så finns det en koppling mellan CardSwitcher klassen till varje sida som finns i programmet. Detta gör att skiftningen mellan det olika sidor sköts i sin egen klass istället för sköta det i App klassen, det gör en även utökningar simpla eftersom man enbart behöver lägga till sidan man vill ha i App klassen samt sätta ett namn på sida så CardSwitcher vet vilken sida den ska byta till. Varje sida som läggs till behöver ta in Cardswitcher klassen i konstruktör för att kunna byta till en annan sida, där man vill kalla på switchTo funktionen vilket sedan sköter skiftningen. Varje sida har även en koppling till Database klassen vilket sköter alla operationer som har med databasen att göra, detta eftersom varje sida har olika saker som den vill begära av databasen. Vilket också visas i UML-diagrammet är att varje sida implementerar från Page, som har metoden switchTo vilket byter sida inom programmet. Varje sida implementerar från Page gränssnittet eftersom det ska finns ett flöde i programmet så man inte kan komma in på någon återvändsgränd. Detta gör det lättare för vidareutveckling.



*UML Diagram för de olika kontroller som används vid temporär lagring av information*

I programmet finns även olika Controllers (AdminController, UserController, LoanController) som är en del MVC design pattern vilket kan användas för skicka data mellan olika sidor. Varje Kontroller klass innehåller en modell och and view, där modellen tar in data och själva klassen håller datan tills den ska visas i viewern. Detta gör att själva sidorna behöver inte hålla reda på information själv utan bara skicka den och sedan visa den. Varje Kontroller implementerar även gränssnittet Controller som innehåller funktionen updateView, vilket gör att den view som är satt i den kontrollern uppdaterar information i viewn. Varje kontroller har även en in Detta gör att varje kontrollern måste ha en funktion som uppdaterar den specifika viewn i kontrollern.

## 6.2.2 Programflöde

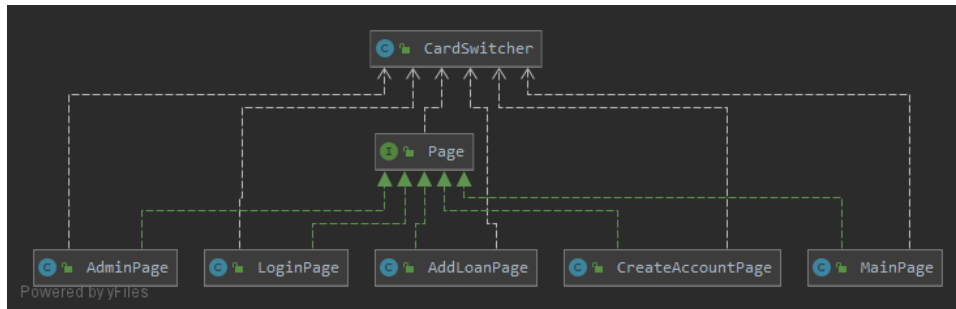
Flödet i programmet är att programmet körs från klassen App där alla sidor intieras samt även alla controllers, detta eftersom bland annat så behöver sidorna ta in CardSwitcher klass i konstruktorn samt att sidorna följer inte en linja utan man kan hoppa till samma sida på olika sätt. Efter App klassen körts så möts man av loginsidan (LoginPage klassen) där man välja att logga in med ett befintligt konto eller skapa ett nytt vanlig konto eller ett admin konto. Om man väljer att skapa ett nytt konto så kommer man till skapa kontosidan (CreateAccountPage) där man då kan skapa ett konto, ens indata valideras samt sparas i en databas. Om man instället väljer att logga in med ett vanlig registrerat konto så kommer man till den så kallade huvudsidan (MainPage) där man se över sina lån samt gå vidare till att lägga till ett lån. Från huvudsidan kan man även välja att logga ut kommer man tillbaka till loginsidan. Om du istället väljer att logga in som en adminanvändare så kommer du komma till huvudsidan för adminanvändare (AdminPage) vilket ger en tabell där alla de ordinarie användarna kan visas, som adminanvändare kan du även redigera de ordinarie användaras inloggningsuppgifter.

### 6.2.2.1 Huvudidé

Flödet i programmet krestar runt klasserna User och Loan, där basen är användare skapar sig en användare för att sedan lägga till lån som använder sedan har sparade och kan hantera dess i huvudsidan av programmet (MainPage). Extra funktionalliteter som har lagts till är bland annat att adminanvändare kan skapas för att kunna hantera de redan existerande användare genom att lägga till/ta bort samt ändra användare. Eftersom flera användare ska kunna existera så måste någon form av lagring ske, programmet började med att ha en json fil som lagringsfil för användare samt deras lån men byttes sedan ut mot en databas. En idé som grundande för programmet var även att användare aldrig ska kunna hamn i någon sorts återvändsgränd utan alltid kunna röra sig "fritt"

mellan de olika sidorna.

### 6.2.2.2 Övergången mellan sidor

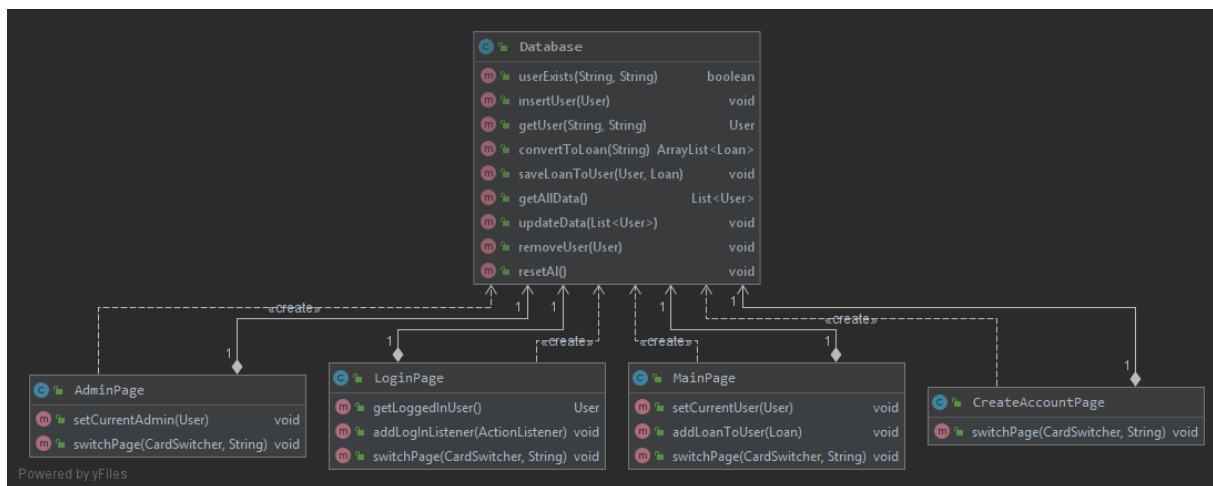


Övergången mellan de olika sidorna sköts av en cardlayout switcher. Där cardlayout fungerar som följande att JFrame som håller all grafisk information i programmet fungerar som container och varje sida som är en JPanel fungerar som kort som kan visas ett åt gången i containern. Detta gör att borttagning av sidor när användaren vill gå vidare inte behövs eftersom det bara kan visas ett kort åt gången i stacken av kort som finns i containern. Denna övergång är i sin egen klass eftersom då kan den tas in i konstruktorn för varje sida för att därefter användas fritt inom sidan för att gå fram och tillbaka. Även Page interfacen har en koppling till CardSwitcher klassen eftersom varje sida har ett krav från Page att ha en switchPage funktion vilket gör det möjligt att kalla på switchTo funktion i CardSwitcher som byter sida.

### 6.2.2.3 Lagring från textfil till databas

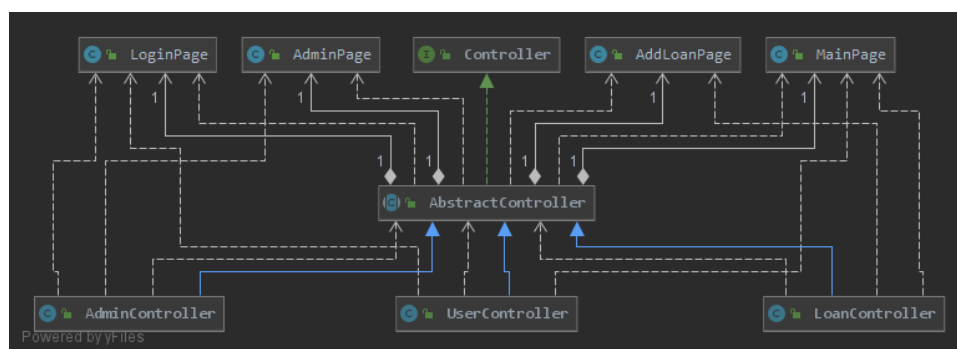
Programmet började med att ha en textfil för lagring för användare samt deras lån, flera olika lösningar finns hur man kan lagra sådan typ data i en textfil. Jag valde att använda .json fil vilket gjorde att jag kunde använda Gson java biblioteket vilket gör att överföring från en Java klass till text och vice versa väldigt enkelt. Där jag valde att använda en HashMap för både användarna samt deras en HashMap för deras lån detta på grund av HashMap instant lookup time vilket gör det väldigt lätt att söka igenom en stor textfil efter en specifik användare. En HashMap fungerar likt Pythons dictionary där man har ett sorts id till ett visst värde där användarens unika id är hashmap nyckel och själva användaren är värdet i form av User klassen. Där User klassen har en HashMap bestående av deras lån i samma form som tidigare där unika id:et till lånet är bundet till själva Loan klassen och alla dessa instans variabler. Detta skapade många olika lager abstraktioner vilket i slutändan gjorde sökning mer komplicerad än vad den gjorde nytta vilket gjorde att HashMapen byttes ut mot ArrayLists vilket gjorde det lättare att loopa igenom listan för att hitta den sökta användare samt deras lån. Vid många användare samt många lån gjorde det svårt att hålla skrivning och läsning konsekutiv eftersom vid varje ändring i textfilen så var en huvud variabel som höll alla data i textfilen tvungen att uppdateras vilket skapade onödigt många läsningar från textfilen.





Detta gjorde att textfilen byttes ut mot en SQLite databas där indexering av användarna och relationen mellan användare och lån sköttes av databasen med hjälp av en one-to-many relation. Vilket betyder att en användare kan ha flera lån med hjälp av en foreign key som kopplas till id av lånet som sedan i sin tur har ett index i lån tabellen. Detta gör att skrivning, ändring och radering av användare samt deras lån simplificeras eftersom SQLite också har en constant lookup time, vid ändring är det också bara just den raden vid användare som ändras och sparas vilket tar bort onödiga läsningar samt skrivning. Vid en borttagning av en användare som har sparade lån som är foreign programmerad till Cascade: ta bort användares lån också annars skulle det skapas en foreig key conflict eftersom man inte kan radera en användare som den har foreign keys kopplade till sparade lån. Vid adminsidan finns även möjligheten att lägga till användare, ändra existerande information och ta bort användare som visas i en tabell på sidan. Vid en ändring i tabellen så finns det extra åtgärder som gör det omöjligt för en admin användare att skicka in fel data som ska lagras detta eftersom vid radering av användar innan man sparar sina ändring så finns det i funktion saveData så finns det två listor en lista som håller alla id:en som är uppdaterade vilket ger adminen en överblick på vilka rader som har ändrats dessutom en lista som går igenom alla emails som har ändrats. Emailen valideras med hjälp av Validator klassens funktion validateEmail och om några av dessa emails inte är giltiga dessa det även ut en meddelande iform av en JDialog där admin får fortsätta trots felaktiga emails eller gå tillbaka och ändra de felaktiga. Funktion getNoDuplicate (AdminPage) gör även att den går igenom alla rader i tabellen och lägger till alla rader där någon slags information har ändrats och det är sedan denna lista som kommer att skickas till databas funktionen updateData (Database) vilket gör så att onödig data inte uppdateras där det inte skett någon ändring. Detta eftersom vid stora antal användare och många ändring hos adminsidan kan göra att programmet sakats ner eftersom stora mängder data skrivs i onödan.

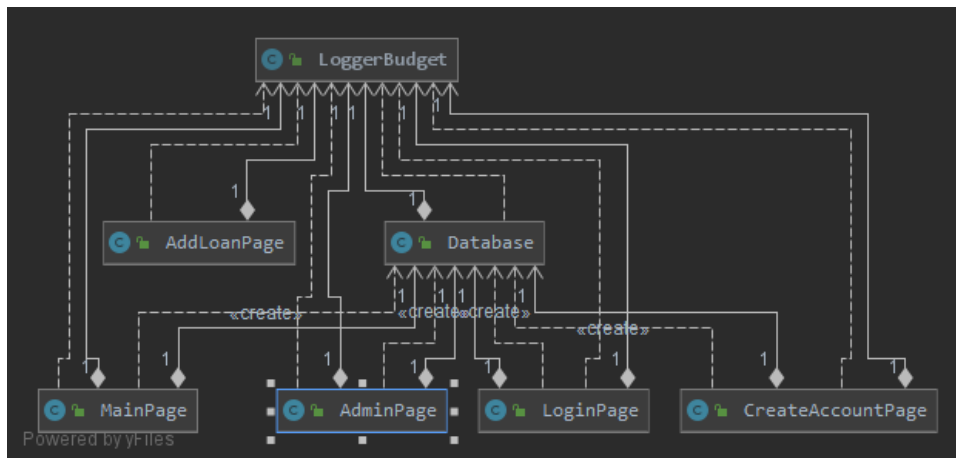
#### 6.2.2.4 Lagring av temporär data (MVC Controllers)



Temporär data behövs i detta program eftersom vid en inloggning så byter vi sida vilket gör att den datan som skrevs in i den första sidan måste föras vidare till den andra sidan som en parameter i konstruktorn på den andra sidan. Men om alla sidorn inte

behöver någon information för att skapas så är inte alla sidor kontinuerliga vilket gör att en expanderings eller utökning blir svårare. Detta är kortfattat löst med att ha en mellan klass som håller den information tills den ska användas som t.ex. från inloggningssidan till huvudsidan där information om vilken användare som är inloggad finns eftersom en användare ska kunna lägga till lån till sig själv och inte till någon annan. Lösning kallas MVC design pattern och används ofta vid design och interaktionsaspekten av hemsidor, där en så kallad kontroller tar in både t.ex. LoginPage (the model) och MainPage (the view) som parametrar i konstruktor vilket gör att kontroller tar in vilken användare som har loggat in när logiknappen har tryckts men eftersom den även kommer åt MainPage's funktioner så kan uppdatera sidan om vilken användare som är inloggad. Vilket gör att sidorna separeras efter funktion då inloggningssidan enbart har hand om ren data och inte bryr sig om hur MainPage uppdateras och det samma med MainPage där den enbart bryr sig om att uppdateras och inte vilken data den får. Detta gör att sidorna inte får fler funktioner än vad som är nödvändigt och skapar extra krångligheter. Det gör att det kan finnas flera views som kan uppdateras separat från modelsidan vilket gör det till en funktionell expanderings trots att det läggs till ett extra lager av abstraktion.

### 6.2.2.5 Logging

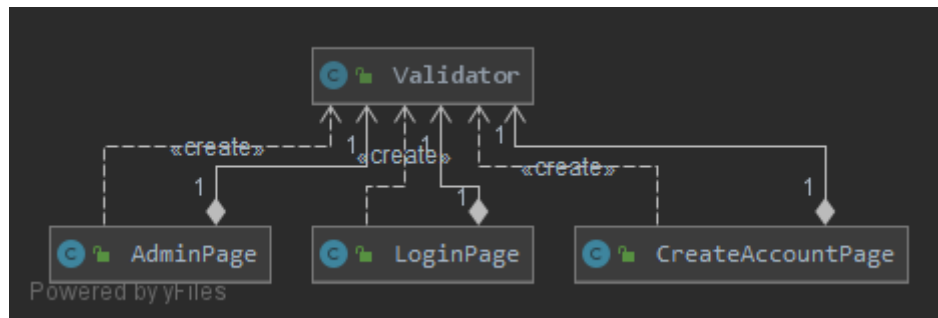


Logging är en viktig aspekt i ett program eftersom om programmet ska användas på en kommersiell nivå så kör inte användare programmet via en IDE detta gör att den inte sker de felmeddelanden som skickas till konsolen. De fel som visas i konsolen är bland annat från catch delen av en try-catch sats där t.ex. stacktracen printas till konsolen vilket gör det lättare för en utvecklare att lokalisera felet men inte för en användare. Där används en logfil som loggar dessa fel i en textfil istället eftersom då kan användaren komma åt felmeddelanden som t.ex. vid felaktigt inloggning osv.

Loggerklassen (LoggerBudget) skapas i App klassen vid programmets start och skickas vidare som inparametrar till varje sida, där den sidan instansieras i sidans konstruktor. Om de skapades en Loggerklass vid varje sidan skulle en ny logfil skapas vid varje sidbyte vilket skulle göra att den gamla filen skrivs över vilket trivialt gör en loggfil värdelös. I konstruktor på Loggern så laddas det in en config fil till hur logfilen ska bete sig det sätts en en FileHandler som talar om vart logfilen ska lagras och även formaterare vilket gör logfilen mer lättläst. I Loggerklassen så finns det även två ytterligare funktioner disableConsoleOutput och logMsg, där disableConsoleOutput tar bort information om de grafiska komponenterna samt databasen som annars skulle överflöda både logfilen och konsolen. LogMsg gör det möjligt att logga meddelande till logfilen från vilken klass som helst bara den instansieras först. Det är inte bara felmeddelanden som visas i logfilen utan även när en användare loggar in, vilken typ av användare som loggar, när en admin ändrar i tabellen över användarna samt när eventuella lån skapas. Detta eftersom användare ska kunna följa sin gång igenom

programmet och se vem som är inloggad och vad användare har gjort under sin gång igenom programmet. Det gör även lättare för en admin då den kan gå tillbaka i logfilen och se när användare tog borts, ändrades eller las till.

### 6.2.2.6 Validering



Validering i ett program som innefattar kontoskapning och inlogging eftersom det motverkar att bland annat felaktiga emails användas även att rätt användare loggas in sina inloggninsuppgifter. Validator klassen är det som sköter alla användar input validering i programmet, validering skedde först separat i klassen där den behövdes men vart eftersom programmet utvecklas så behövdes samma kod för validering detta gjorde att den extraherades till sin egen klass vilket gör det både lättare för expanderings då extra lager av validering kan läggas till i klassen för att sedan användas vid de olika sidorna. Just nu finns det tre funktioner tillgängliga: validateEmptyInput, validateIsString och validateEmail, som sagt så finns det utrymme för expanderings där lösenords restriktioner bland annat kan läggas till. Först så hade jag även en logger i validatorklassen för att ta bort repetativ kod från sidorna, men eftersom jag vill att de meddelanden som skrivs till logfilen så vara så pass anpassade så att användaren lätt kan följa dom valde jag att flytta dom till sidorna så att användare ser vart och när meddelandet skrevs.

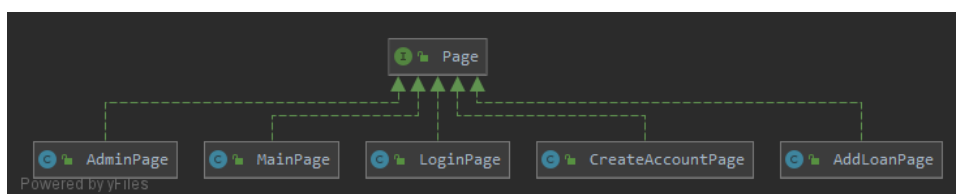
### 6.2.2.7 Abstrakta klasser & Interfaces

#### Abstrakta klasser

De abstrakta klasser som finns är AbstractUser och AbstractController. AbstractUser innehåller information som varje User alltid kommer att ha därför är de instansvariabler extraherade till sin egen klass eftersom varje användare kommer alltid ha en email och ett lösenord detta gäller även för get-funktioner för dessa variabler. Det finns även en tom konstruktor eftersom det finns en tom konstruktor i User klassen som används vid utloggning då den inloggade användaren sätts till en tom användare vid utloggning.

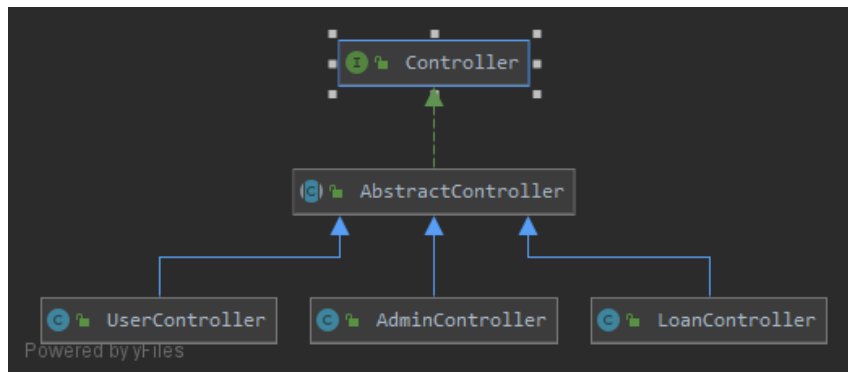
Tillskillnad mot AbstractUser så har AbstractController tre olika konstruktörer eftersom det finns tre kontroller för att de tar in olika sidor som inparameterar i konstruktorn. Det kommer alltså att vara olika modeller och olika views beroende på vilket kontroller som används t.ex. AdminController som har LoginPage som modell och AdminPage som view medan LoanController har AddLoanPage som modell och MainPage som view.

#### Interfaces



De interfaces som finns är Page och Controller, där interface sätter krav på klasserna

måste ha vissa funktioner och parametarer. Page interface sätter ett krav på att varje sida måste ha en switchPage funktioner det vill säga den funktion som byter mellan sidorna, den krävs eftersom användaren aldrig ska fastna i programmet utan kunna röra sig fritt. Funktionen tar in CardSwitcher klassen samt en sträng på den nya sidan, CardSwitcher har i sin tur en funktion switchTo vilket tar in en sträng som bestämmer vilket sida som ska visas. Strängarna kopplas till sidorna i konstruktorn i App klassen som krävs för att en CardLayout ska veta vilken sida som bytas till.



Controller interfacen har funktion `updateView` som sedan implementeras separat i varje kontroller klass, eftersom `view:n` ska kunna ta emot den datan som modellen skickar. Detta kan bland annat vara vilken användare som är inloggade men även vilket lån som ska sättas till vilken användare. Funktionen `updateView` kallas på i den interna lyssnar klassen som i sin tur sätter den lämpliga datan som skickas från modellen och sätter den i `view:n`.

## 7. Användarmanual

\*\* Till person som ska köra programmet för bedömning vill jag gärna att ni säger till i förväg när i tänkte gå igenom programmet eftersom jag måste sätta igång servern som håller databasen. Databasfilerna i mappen *databasefiles* där bedömmaren måste ha XAMMP nedladdat och sedan skapa en databas med namnet `tddd78` för att sedan importera filerna `loan.sql` samt `user.sql` som kommer med några inlaggade användare med existerande lån. Detta eftersom databasen körs lokalt eftersom jag inte kunde hitta någon hosting site som var lämplig. \*\*

Programmet körs sedan med att starta App klassen, genom att högerklicka i Enum klassen Utility. Den första sidan ni kommer att mötas av är inloggningsidan där ni kan välja att logga in med en existerande användare eller att gå vidare till att skapa en egen användare via *skapa konto* knappen.

Då kommer ni att komma till kontoskapningsidan där ni kan välja att skapa en “vanlig” eller en admin användare. Tänk på att namnet inte får innehålla siffror samt att email måste vara en riktigt email. Efter ni har skapat ett konto så kommer ni komma tillbaka till inloggningsidan

där ni loggar in på eran nyskapade användare.

Om ni har skapat en “vanlig” användare och loggat in kommer ni kommer in till huvudsidan där ni kan se era skapade lån. Om ni vill skapa ett nytt lån så klickar ni på *lägg till lån* knappen vilket sedan där till sidan där nya lån kan skapas, tänk på att start och slut datum är satta till dagens datum och slutdatumet måste ligga före i tid än startdatumet. Om ni vill avsluta och inte lägga till ett lån så klicka på *avsluta*. I huvudsidan kan ni bläddra bland era lån genom att klicka på de olika tabbarna som finns överst i det stora fältet. Om ni vill logga ut så klicka på *logga ut*.

The screenshot shows a login page titled "\*BUDGET\*". It contains three input fields for "Namn:", "Email:", and "Lösenord:". Below the password field are two radio buttons labeled "Vanlig" (selected) and "Admin". At the bottom, there is a "Skapa konto" button, a link "har du redan ett konto?", and a "Logga in" button.

The screenshot shows the main dashboard titled "\*BUDGET\*". At the top, there are buttons for "Lägg till lån" and "Logga ut". Below these are tabs for "testdb2", "Test", "Test213", and "Testawdwd". The "testdb2" tab is active, showing a form with fields for "14 nov. 2000", "11 apr. 2020", "Lånebelopp: 100", "Ränta: 10.0", "Amortering: 1", and "Beskrivning: testdb".

The screenshot shows the "Lägg till lån" form. It has a title "Lägg till lån" and an "Avsluta" button. The form contains several input fields: "Rubrik: Test", "Startdatum: 27 maj 2020", "Slutdatum: 27 maj 2020", "Mängd: 1000", "Ränta(%): 1000", "Amortering(kr): 1000", and "Beskrivning: testdesc". At the bottom, there is a "Lägg till lån" button.

Om ni väljer att skapa en admin användare eller att logga in på en existerande så kommer ni efter inloggningen att mötas av adminsidan som består av ett tomt fält och en rad knappar. För att visa alla existerande användare så klickar ni på *visa användare*, nu kan ni ändra direkt i tabellen med användarnas information samt lägga till en rad med knappen *ny användare* eller ta bort en användare för att sedan välja index på den valda användaren i popupen via knappen *ta bort användare*. När ni är klara med alla ändringar så klickar ni på *spara ändringar* för att spara ändringar ni har gjort till databasen. Om ni har ändrat en email till en felaktig email och försöker spara kommer ni mötas av en popup som berättar att det finns felaktiga emails i tabellen där kan ni välja att fortsätta ändå eller att klicka *nej* och ändra till en giltig email. Om ni vill logga ut så klickar ni på *logga ut*.


**\*BUDGET ADMIN\***

ID	Namn	Email	Lösenord	Användartyp	Antal Lån
1	calle	test@gmail...	testlösen	ORDINARY	6
8	calletest	no@gmail...	nolosen	ORDINARY	4

**Ta bort användare**

Välj ett konto med det id du vill ta bort

**Felaktiga Emails**

 Det finns felaktiga emails vid id: [1].  
Vill du fortsätta?