|  |  |
| --- | --- |
| Anja Andersen siger farvel til håndbold | Sport | DR  HåndboldApp  Antal tegn: XXX | Henrik flensborg  johannes kizach  martin farsø |

Indholdsfortegnelse

[1 Indledning (Henrik) 2](#_Toc188441969)

[2 Metode 2](#_Toc188441970)

[2.1 Gruppekontrakt (Henrik) 2](#_Toc188441971)

[2.2 Azure DevOps (Henrik) 2](#_Toc188441972)

[2.3 Vandfaldsmodellen (Henrik) 2](#_Toc188441973)

[2.4 3-lags-arkitektur (Henrik) 3](#_Toc188441974)

[2.5 Designvalg / kodestandard (Henrik & Johannes) 3](#_Toc188441975)

[2.6 Sharepoint / Teams (Henrik) 4](#_Toc188441976)

[2.7 ChatGPT (Henrik) 4](#_Toc188441977)

[3 Kravspecifikation 5](#_Toc188441978)

[3.1 Krav til databasen (Henrik) 5](#_Toc188441979)

[3.2 Krav til programmet (Henrik) 5](#_Toc188441980)

[3.3 Problemidentifikation (Henrik) 5](#_Toc188441981)

[3.4 Use Case-diagram (Henrik) 5](#_Toc188441982)

[4 Design 6](#_Toc188441983)

[4.1 Databasedesign (Johannes) 6](#_Toc188441984)

[4.1.1 Tabeller 6](#_Toc188441985)

[4.1.2 Normalisering 7](#_Toc188441986)

[4.2 Interfacedesign (Martin) 8](#_Toc188441987)

[5 Implementering 9](#_Toc188441988)

[5.1 Databasen og stored procedures (Johannes) 9](#_Toc188441989)

[5.2 GUI-implementering (Martin) 11](#_Toc188441990)

[5.3 Controller-klasserne (Martin) 13](#_Toc188441991)

[5.4 Sekvensdiagram – Gem kamp (Johannes og Martin) 14](#_Toc188441992)

[5.5 DButils (Henrik) 15](#_Toc188441993)

[5.6 Import/eksport (Johannes) 15](#_Toc188441994)

[5.7 Klassediagram (Martin) 16](#_Toc188441995)

[6 Afprøvning (Johannes) 16](#_Toc188441996)

[7 Konklusion og refleksioner (Johannes og Martin) 17](#_Toc188441997)

[8 Bibliografi 18](#_Toc188441998)

[9 Bilag 19](#_Toc188441999)

[9.1 Kanbaneksempel 19](#_Toc188442000)

[9.2 Gruppekontrakt 20](#_Toc188442001)

# Indledning (Henrik)

I rapporten gennemgår vi processen med at udvikle en prototype af et program, der skal håndtere registrering af håndboldkampe. Prototypen er designet omkring en 3-lags-arkitektur med fastlagte krav: JavaFX som præsentationslag, Java som forretningslag og MSSQL som datalag. Den er udviklet til at opfylde samtlige eksamenskrav, såsom registrering af mål og udvisninger, stillingen i ligaen, import og eksport af csv-filer m.m. Rapportens opbygning følger strukturen i vandfaldsmodellen (se figur 1 i metodeafsnittet "Vandfaldsmodellen") og indebærer en systematisk gennemgang af faserne: kravspecifikation, design, implementering og afprøvning. Spørgsmålet om vedligeholdelse behandles særskilt og diskuteres i rapportens refleksionsafsnit.

# Metode

## Gruppekontrakt (Henrik)

For at sikre projektets effektivitet startede vi med at definere en gruppekontrakt (se bilag 9.2). I kontrakten aftalte vi at man skal overholde de indgåede aftaler, løfte sin del af projektet og kommunikere tydeligt, hvis der opstod afvigelser fra det planlagte. Derudover er det essentielt, at alt, hvad der afleveres, er blevet gennemgået og forstået af alle gruppemedlemmer.

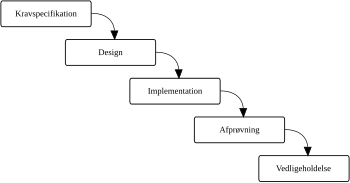
## Azure DevOps (Henrik)

Tidligt i planlægningsprocessen blev vi enige om at anvende Azure DevOps som vores udviklingsmiljø med brug af Azures git-repository. En af de andre afgørende faktorer var, at Azure DevOps også inkluderer et indbygget Kanban-board, som giver et klart overblik over arbejdsopgaver og deres status (se bilag 9.1).

Vi brugte også wiki-funktionen til at tilføje beskrivelser af alle SQL-procedures for at alle kunne se præcis hvilke stored procedures der var implementeret, og hvilke parametre de hver især tog. Vi brugte også wikien til at dokumentere vores git-protokol.

## Vandfaldsmodellen (Henrik)

Vi valgte at udføre projektet med vandfaldsmodellen (Adetokunbo AA & Adenowo, 2013), da modellen passer godt til eksamensopgaver, hvor kravspecifikationerne er tydelige og fastsatte. Eksamensopgaven fungerer som et kravdokument, der indeholder alle nødvendige oplysninger, og derfor er der ikke behov for at tage højde for fleksibilitet eller ændringer i kravene. Projektets opstart bestod derfor af planlægning, brainstorming, udarbejdelse af Use-Case- og ER-diagram samt ideer til kodearkitektur og database design.



Figur Vandfaldsmodellen

## 3-lags-arkitektur (Henrik)

For at opfylde eksamenskravene blev vores prototype opbygget med en 3-lags-arkitektur (Christensen, 2011), der består af følgende lag:

* **Præsentationslag (GUI)**: Håndterer brugerinteraktion og viser eller sender data via et grafisk interface.
* **Applikationslag (Business Logic)**: Indeholder applikationens logik og fungerer som bindeled mellem GUI og databasen.
* **Datalag (Database)**: Ansvarligt for databaseinteraktion via SQL-forespørgsler og stored procedures.

Logikken er derfor simpel: Præsentationslaget kommunikerer med applikationslaget for at hente eller sende data. Applikationslaget fungerer som et bindeled, der både håndterer kommunikationen mellem præsentations- og datalaget og konverterer data til objekter eller lister. Slutteligt er datalaget, som står for opbevaring, hvorfra data hentes og gemmes.

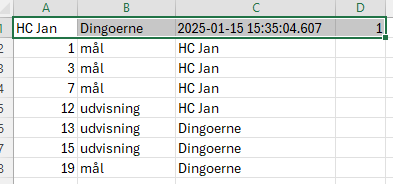
## Designvalg / kodestandard (Henrik & Johannes)

For at sikre en effektiv kodeudvikling aftalte vi både kodestandarder for navngivning og den logiske struktur af vores applikationslag. Kodekommentarer i al kode og brugerinteraktion i GUIen skulle være på dansk, mens navngivning af metoder og klasser i Java skulle være på engelsk og følge camelCase-princippet.

Med vores designvalg ønskede vi at skabe to overordnede utility-klasser: DButils og CSVutils. Begge indeholder statiske funktioner som kan anvendes i de forskellige kontrolklasser uden at skulle oprette en instans af utility-klasserne.

DButils skulle håndtere forbindelsen mellem applikationslaget (som håndterer logikken og processerne i applikationen) og datalaget (som gemmer og henter data fra databasen). DButils indeholder derfor funktioner til at oprette forbindelse til databasen ved hjælp af loadJdbcDriver og openConnection, for at tilgå de nødvendige CRUD-operationer med prepared statements.

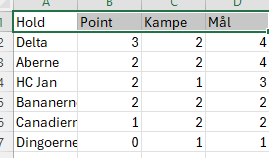
CSVutils indeholder tre funktioner til at importere og eksportere data. Importfunktionen, *readCSV()*, opretter en ny kamp i databasen baseret på en csv-fil som selvfølgelig skal organiseres på en bestemt måde for at den kan læses og genkendes som en ny kamp af programmet.



Figur kamprapportcsv-filens struktur

Den første linje er dedikeret til data om kampen (hjemmehold, udehold, kampstartstidspunkt, resultat) og de følgende linjer er de begivenheder der fandt sted i kampen (tidspunkt, begivenhed, hold).

Den ene eksportfunktion, *writeCSVGameReport()*, udskriver en csv-fil der nøjagtig ligner den man skal bruge for at importere. Den anden funktion, *writeCSVScore()*, udskriver stillingen i ligaen ligesom den vises i GUIen.



Figur Stillingcsv-filens struktur

## Sharepoint / Teams (Henrik)

Vi har delt vores eksamensrapport via SharePoint, da det giver os mulighed for at arbejde på rapporten samtidig i realtid. Samtidig har vi haft adgang til Teams' chat- og callfunktion, hvilket har sikret en løbende og effektiv kommunikation gennem hele projektet.

## ChatGPT (Henrik)

I vores eksamensprojekt har vi anvendt ChatGPT som en tutor til at besvare spørgsmål og guide os gennem udfordringer, som man normalt ville søge svar på via Stack Overflow. Det har bidraget til hurtigere problemløsning og en bedre forståelse af komplekse emner.

# Kravspecifikation

## Krav til databasen (Henrik)

1. Hold skal kunne oprettes.
2. Kampe mellem hold skal kunne oprettes.
3. Mål og udvisninger skal registreres med tidspunkt og hold.
4. Kamprapporter skal kunne vises.
5. Stillingen i ligaen skal kunne vises.

## Krav til programmet (Henrik)

1. **Brugergrænseflade:**

JavaFX GUI.

1. **Integration:**

Javakode, der forbinder databasen og GUI.

1. **Kampfunktioner:**

Kampen skal kunne pauses og genoptages.

1. **Import/eksport:**

En kamprapport skal kunne importeres og eksporteres som CSV-fil.

Stillingen i ligaen skal kunne eksporteres til en CSV-fil.

1. **Opdatering:**

Mulighed for at opdatere og slette hold.

## Problemidentifikation (Henrik)

I vores arbejdsproces identificerede vi flere potentielle komplikationer, der kunne påvirke udarbejdelsen af prototypen. Den første problemstilling var at designe en funktionel database, der ikke blot kunne håndtere de nødvendige datamængder, men også opfylde kravene om effektiv og dynamisk datahåndtering.

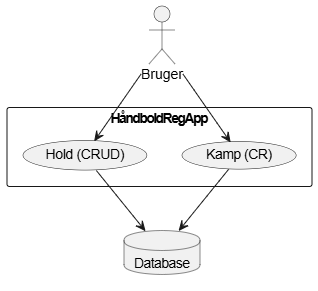
Fx medfører kravet om at et hold skal kunne slettes, at vi skal fjerne hold fra ikke bare tabellen hold, men også de andre tabeller hvor hold indgår som fremmednøgle. Derfor skal vi forholde os til om vi vil slette dem manuelt, tillade null-værdier, eller bruge kaskadermekanismen i SQL.

I forhold til at udvikle klasserne DButils og CSVutils var hovedproblematikken at sikre en problemfri kommunikation mellem databasen og brugerfladen. Vi tænkte at det ville blive klarere i løbet af implementeringen.

## Use Case-diagram (Henrik)

Med Use Case-diagrammet illustrerer vi, hvordan brugeren interagerer med prototypen "HåndboldRegApp".

Applikationen giver brugeren mulighed for at håndtere kampe ved at oprette og læse information (CR). Derudover har brugeren fuld kontrol til at oprette, læse, opdatere og slette data for hold (CRUD). Disse funktioner er direkte forbundet til databasen, som prototypen anvender til at gemme og hente data.



Figur Use Case-diagram

# Design

## Databasedesign (Johannes)

### Tabeller

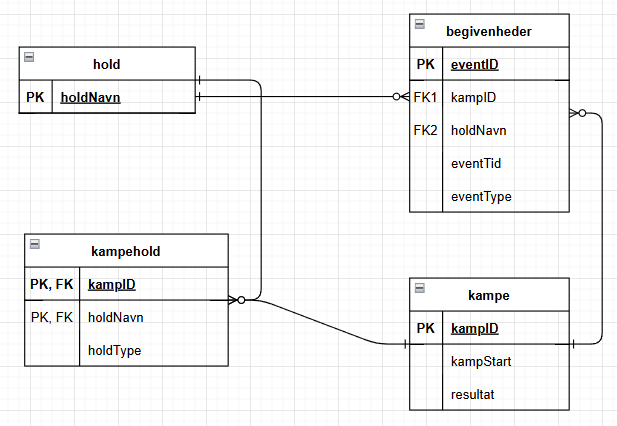
Databasen skal indeholde 4 tabeller: hold, kampe, kamphold og begivenheder.

**Hold** er blot en liste over holdene i ligaen.

**Kampe** indeholder et ID, tidspunktet for kampens begyndelse, samt et resultat (0, 1 eller 2 som indikerer henholdsvis uafgjort, hjemmeholdet vandt, og udeholdet vandt).

**Begivenheder** er en liste over de registrerede mål og udvisninger (eventType), hvor hver begivenhed har et ID, et registreret tidspunkt (det sekund begivenheden fandt sted), en fremmednøgle som viser i hvilken kamp begivenheden fandt sted, samt en anden fremmednøgle som viser hvilket hold der er ansvarlig for målet/udvisningen.

**Kamphold** sikrer at der ikke er en mange-til-mange-relation mellem hold og kampe (to er mange), og har en sammensat primærnøgle som består af de to fremmednøgler: holdNavn (fra hold) og kampID (fra kampe). Derudover indeholder kamphold information om hvorvidt holdet i rækken var hjemmehold eller udehold i den relaterede kamp.



Figur ER-diagram

### Normalisering

Der er ikke nogen transitive afhængigheder mellem de attributter som ikke er primærnøgler i tabellerne, og ikke-nøgler er afhængige af nøglerne.

I tabellen hold er det klart at der ikke er afhængigheder (eneste attribut i tabellen er primærnøglen). Ligeledes er der i kamphold kun en attribut som ikke er en del af primærnøglen, og holdtypen (hjemme- eller udehold) kan ikke forudsiges af anden information.

I kampe er de to attributter uafhængige af hinanden – der kan starte kampe samtidigt, og kampe kan have samme resultat, så starttidspunkt og resultat er ikke nok til unikt at identificere en given kamp, og følger ikke af hinanden – men følger af *kampID*, primærnøglen.

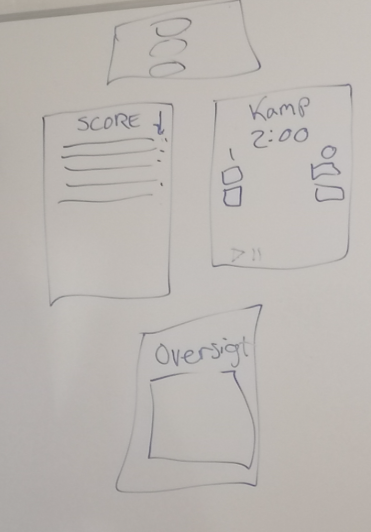
I tabellen begivenheder er ingen af attributterne som ikke er primærnøglen, afhængige af hinanden: et mål/en udvisning kan forekomme på alle tider, i alle kampe, og udføres af alle hold. Omvendt kan primærnøglen, *eventID*, alene unikt identificere de andre attributter.

Ét hold kan knyttes til mange begivenheder, én kamp knyttes til mange begivenheder. Og endelig kan en kamp knyttes til flere kamphold. Så der er kun en-til-mange relationer i databasen, og i de tabeller som indgår som mange-delen i de relationer, er der en fremmednøgle fra den tabel der indgår som en-delen. Eftersom der heller ikke er transitive afhængigheder, så må tabellerne være i tredje normalform (Connolly & Begg, 2005).

Vi vil bruge stored procedures i databasen for alle de funktioner vi skulle bruge, dels af sikkerhedshensyn, og dels fordi det ville gøre DButils-funktionerne ensartede og derfor enklere at kode og vedligeholde.

## Interfacedesign (Martin)

Interfacet er bygget op på baggrund af Gestaltlovene, med fokus på nærhed, lighed, lukkethed, forbundethed og loven om figur og baggrund (Kirkegaard, 2025). Alle elementer har en klar funktion og er grupperet tæt på sammenhængende elementer. Ligeledes er data også sat i kolonner, hvilket skaber en visuel sammenhæng mellem de forskellige kolonner. Data skal vises i ListViews og kan dermed ikke manipuleres eller ændres efter import fra databasen. Brugen af ListView gør yderligere at sorteringen af data ikke kan gå tabt. Der bruges BootstrapFX som CSS-framework til tekst og knapper, hvilket bidrager til ligheden mellem vinduerne i programmet.



Figur Præliminært design af GUI

# Implementering

## Databasen og stored procedures (Johannes)

Tabellerne i databasen blev sat op som planlagt (se ER-diagrammet i figur 5), og værdier blev indsat så vi kunne teste funktioner på data. For at lette sletning af hold brugte vi *on delete cascade* der hvor hold er fremmednøgle og der hvor kamp er fremmednøgle. Fordi når man sletter et hold, så skal de rækker i kamphold hvor holdet indgår også slettes, og dermed skal man jo også slette den række i kamphold hvor modstanderholdet står. Og endelig skal man jo slette de rækker i begivenheder som hører til de kampe det slettede hold deltog i. Det betyder at man skal identificere de kampe holdet deltog i, og så slette alle de steder hvor kampID er fremmednøgle.

*On delete cascade* gør det derfor muligt at have den her forholdsvis enkle stored procedure til at slette et hold fra databasen:

CREATE PROCEDURE removeTeam @navn nvarchar(30)

AS

BEGIN

DELETE FROM kampe

WHERE kampID IN (

kampID

FROM kamphold

holdNavn = @navn

)

DELETE FROM hold WHERE holdNavn = @navn

END;

Tilsvarende bruger vi *on update cascade* for at lette ændringen af et holdnavn.

De fleste stored procedures var forholdsvist enkle at kode (se vedlagte SQL-kode), men *addGame*-funktionen var ret kompleks, og derfor gennemgår vi den her – afsnit for afsnit med forklaringer:

CREATE PROCEDURE addGame @hjemmehold NVARCHAR(30), @udehold NVARCHAR(30), @start DATETIME, @res INT, @jsonData NVARCHAR(MAX)

Funktionen tager fem parametre hvor det femte er en jsonvariabel. Det er en tekstgengivelse af hvad der i java ville være et Array af HashMaps, og vores jsonData har den her struktur:

'[{"eventTid": 1, "eventType": "mål", "holdNavn": "Elefanterne"},

{"eventTid": 2, "eventType": "udvisning", "holdNavn":"Delta"}]'

Fordelen er at vi på den måde kan samle alle en kamps begivenheder i en liste og så indlæse dem i databasen på én gang. En begivenhed har også et kampID som attribut – og hvis vi i stedet skulle indlæse begivenhederne én ad gangen med en anden funktion, så ville vi få en to problemer:

* Begivenheder ville kun kunne indføres efter at en kamp er oprettet
* Før man indfører begivenheder, ville man skulle hente den senest tilføjede kamps kampID

Begge problemer undgås ved at bruge json.

IF (SELECT COUNT(\*) FROM hold

WHERE holdNavn IN (@hjemmehold, @udehold)) < 2

BEGIN

;THROW 50002, 'Enten hjemmeholdet, udeholdet eller begge findes ikke i tabellen hold.', 1;

END;

If-sætningen sikrer at dataindlæsningen ikke finder sted hvis ikke begge holdnavne faktisk er oprettet i tabellen hold. Vi tæller bare antallet af rækker i hold der indeholder de to holdargumenter – det skal jo være 2 hvis begge hold findes.

DECLARE @data TABLE (

eventTid INT,

eventType NVARCHAR(10),

kampID INT,

holdNavn NVARCHAR(30)

);

Her opretter vi en midlertidig tabel (den slettes når funktionen er færdig) som vi senere bruger til at pakke vores jsonData ud i.

INSERT INTO kampe VALUES

(@start,@res);

Den nye række i kampe oprettes med de to argumenter der angiver starttidspunktet og resultatet – kampID oprettes automatisk: markeret med IDENTITY i SQL-koden der opretter tabellen kampe.

DECLARE @nytKampID INT;

SET @nytKampID = SCOPE\_IDENTITY();

Funktionen SCOPE\_IDENTITY() henter netop den sidst tilføjede int i en IDENTITY-attribut. Vi bruger den int-variabel som værdi for en ny variabel nytKampID, som vi skal bruge når vi indlæser data i tabellerne kamphold og begivenheder.

INSERT INTO kamphold VALUES

(@nytKampID, @hjemmehold, 'hjemme'),

(@nytKampID, @udehold, 'ude');

Hjemmeholdet og udeholdet oprettes som to linjer i kamphold – med den nye kamps kampID.

INSERT INTO @data (eventTid, eventType, kampID, holdNavn)

SELECT

JSON\_VALUE(value, '$.eventTid') AS eventTid,

JSON\_VALUE(value, '$.eventType') AS eventType,

@nytKampID AS kampID,

JSON\_VALUE(value, '$.holdNavn') AS holdNavn

FROM OPENJSON(@jsonData);

Her tager vi så data fra vores jsonvariabel og indsætter dem i den midlertidige tabel vi lavede til at starte med. Bemærk at vi igen bruger vores nytKampID-variabel som kampID. De andre værdier får vi med en syntaks der ligner HashMap-syntaks i java. Vi angiver nøglen med *$.nøglenavn*, og værdien hedder belejligt bare *value* når man bruger funktionen JSON\_VALUE().

OPENJSON() er funktionen som parser en jsonliste og derfor gør det muligt at tilgå den strukturerede tekst – med nøgler og værdier.

INSERT INTO begivenheder (eventTid, eventType, kampID, holdNavn)

SELECT eventTid, eventType, kampID, holdNavn

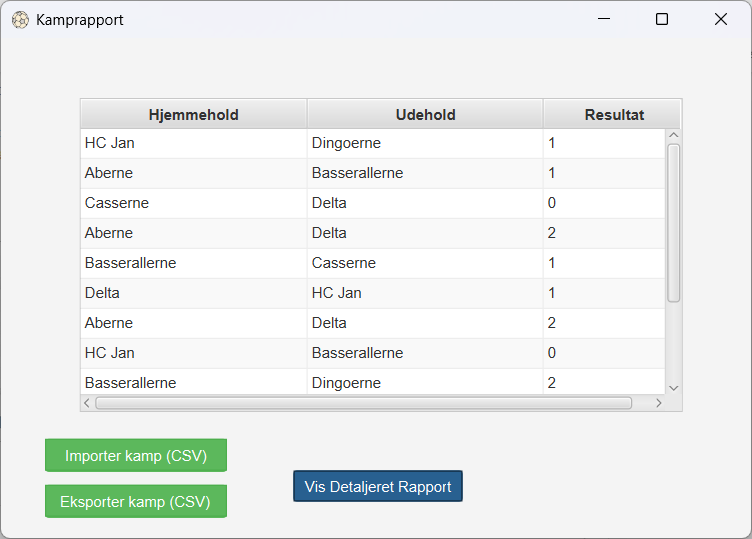
FROM @data;

Det sidste er det simpleste: Vi tager ganske enkelt hele den midlertidige tabel, *data*, og putter dens kolonner ind i tabellen begivenheder. Hver begivenhed får sit unikke ID fordi også her er *eventID* en IDENTITY-kolonne.

## GUI-implementering (Martin)

GUI blev udviklet ud fra de rammer der blev sat under designfasen. Vi implementerede GUI med henblik på at brugeren nemt forstår programmet og ved hjælp af restriktioner, forhindre at fejl forekommer.

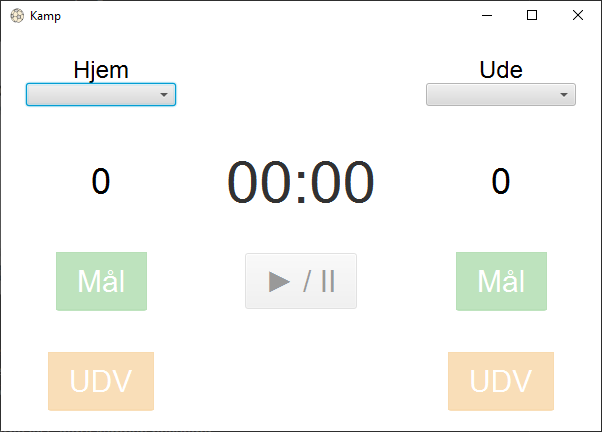
Ved vores GUI til kamprapport og stillingsoversigt bruger vi farvekodning til at holde en rød tråd mellem vinduerne.

A screenshot of a computer

Description automatically generated

Figur Kamprapport og Stillingsoversigt

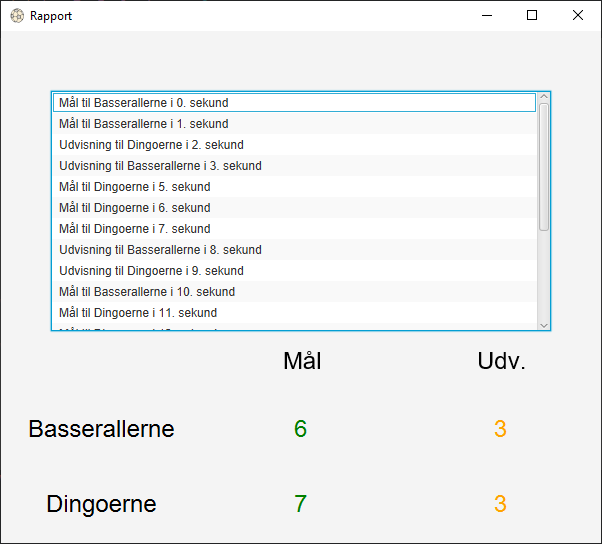
Ved oprettelsen af en kamp er der yderligere restriktioner for at begrænse muligheden for, at brugeren laver fejl. Funktioner bliver løbende låst og åbnet i takt med, at oprettelsen og udførelsen af kampen sker. Valg af hold forbliver åbent indtil kampstart, hvorefter de bliver låst. Det er ikke muligt at starte kampen, hvis ikke begge hold er valgt, eller hvis holdene er ens. Omvendt forbliver mål- og UDV-knapperne låst, indtil kampen begynder.

A screenshot of a video game

Description automatically generated

Figur Kamp inden start og kamp efter start

Brugerinterfacet i vores detaljerede rapport viser en liste over kampens hændelser i kronologisk rækkefølge: mål og udvisninger med tidspunkter for hver hændelse. Nederst vises labels med statistikker for hvert hold, hvor mål (grøn) og udvisninger (orange) er angivet i separate kolonner. Layoutet er enkelt og funktionelt med farvekodning der matcher oprettelsen af en kamp.



Figur Detaljeret rapport

Oprindeligt var vores tanke at bruge ListView til også vise stilling og kamprapport, men formateringen viste sig at være besværlig fordi BootstrapFX låser fonttypen – og for at få kolonner i samme bredde skal man helst bruge en font der har en fast og ens bredde for alle tegn. Derfor endte vi med at bruge TableView. TableView gav nye udfordringer, da det nu var muligt for brugeren at flytte rundt på kolonnerne og tilpasse deres bredde, hvilket vi ikke ønskede. Derudover var der også problemer med sorteringen af data, da TableView tillader brugeren at sortere data, men kun har mulighed for sortere efter én kolonne, hvor vi gerne vil sortere først efter point og dernæst efter mål (fordi det er jo sådan en stilling fungerer).  
  
Vi løste problemet ved at lave en stored procedure, der sorterer data inden vi indsætter den i tabellerne. Derefter låser vi sorteringsmulighederne og omrokeringsmulighederne med de indbyggede metoder fra TableView-klassen. På den måde undgår vi at vores oprindelige sortering af data går tabt, brugeren kan ikke manipulere eller ændre på data og data står nu som vi ønsker.

## Controller-klasserne (Martin)

For at minimere antallet af controllers, var vores udgangspunkt fire controllers, men da vi benytter os af initialize-metoden for at indsætte data i vores tabeller, ville det skabe problemer, hvis viewet ikke indeholdt de tabeller. Programmet blev derfor udvidet til at have kun ét view for hver enkelt controller. Det udvider derfor også behovet for flere controllers, da vi i løbet af programmet har meget små views, som kun har én enkelt funktion (fx til indtastning af nyt holdnavn).

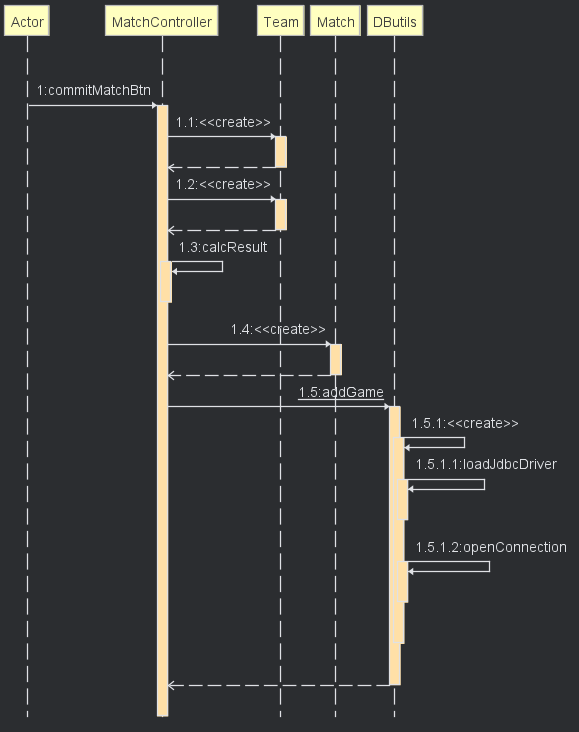
Vi vælger at dykke ned i MatchController og Det

ailedReportController, da de begge er interessante og har unikke funktioner.

MatchController er en central klasse, der styrer logikken og brugergrænsefladen for en kamp. Den administrerer tidshåndtering gennem en integreret Timer og opdaterer kampens visuelle elementer, som holdnavne og scores. Klassen indeholder metoder til at initialisere kampe og opdatere UI-elementer baseret på kampens status. MatchController pakker til sidst kampenes elementer ind i en List<Object>, som kan bruges af DButils til at føje kampen til databasen.

DetailedReportController er interessant pga. måden vi håndterer data på. Vi gemmer ikke scoren på kampene i databasen, kun resultatet, så for at få antal mål og udvisninger ud, skal alle events fra kampene loopes igennem og alt efter type, optælles. Samtidigt bliver de tilføjet til et ListView, så hvert event står i rækkefølge efter tidspunkt.

## Sekvensdiagram – Gem kamp (Johannes og Martin)



Figur Sekvensdiagram for commitMatchBtn

Når man trykker på knappen *Gem kamp* i GUIen (som hedder commitMatchBtn i javakoden), oprettes de to Team-klasser og resultatet beregnes. Derefter oprettes en Match-klasse, og en DButils-funktion kaldes, som gemmer resultatet i databasen.

Som man kan se, illustrerer sekvensdiagrammet hvordan vi blander præsentationslaget og applikationslaget i vores MatchController – oprettelsen af Match- og Team-klasser hører strengt taget til applikationslaget, og det samme gør beregningen af resultatet. Præsentationslaget burde egentlig kun forholde sig til at vise målscoren, og registrere brugeres tryk på diverse knapper. Det der sker bagved, når der trykkes, burde vel så ligge i en anden klasse. Med DButils ligger applikationsfunktionalitet i sin egen klasse – men det kunne oprettelsen af klasser og beregningen af resultatet også have gjort.

## DButils (Henrik)

Formålet med DButils var tydelig: Der skulle implementeres funktioner, der matchede de stored procedures, vi havde kodet i SQL. Målet var at sikre, at applikationen kunne kommunikere med databasen på en effektiv og struktureret måde.

I de tidlige faser af udviklingen valgte vi at implementere metoderne i DButils ved hjælp af HashMaps. Den tilgang blev valgt, fordi HashMaps giver en fleksibel måde at håndtere dynamiske data på, da de indeholder nøgle-værdi par, hvilket i teorien kunne gøre det nemt at gemme og hente resultater fra SQL-forespørgsler uden at skulle oprette flere klasser.

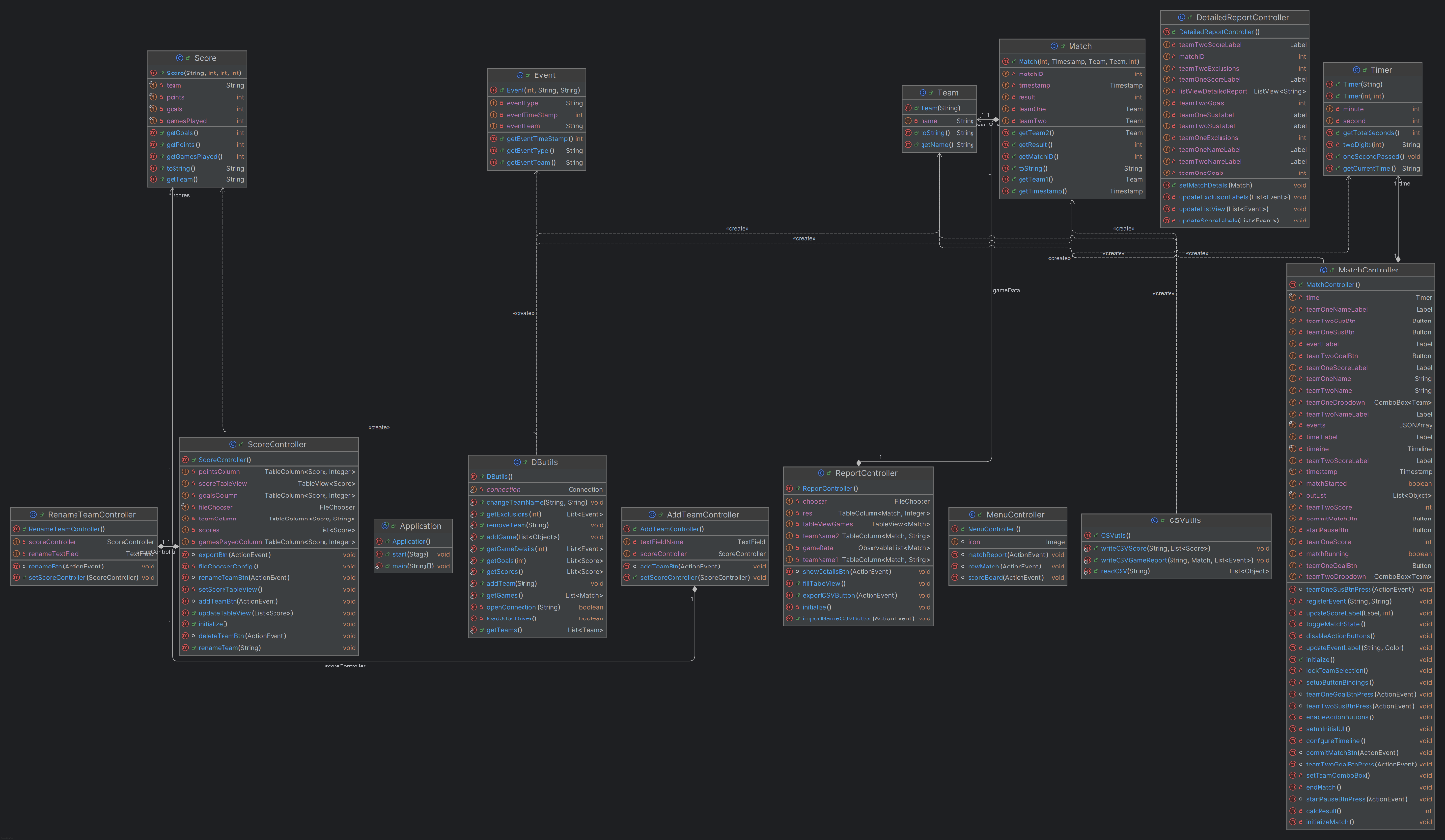
Dog viste det sig at metoden var upraktisk, når det kom til at overføre data til præsentationslaget. HashMap giver kun begrænset information om den faktiske struktur af de data, den indeholder, hvilket gjorde det vanskeligt at arbejde med data i brugergrænsefladen. For at præsentere data på en mere struktureret og forståelig måde i appen, besluttede vi derfor at skifte til en mere traditionel objektorienteret tilgang.

Ændringen medførte, at vi i stedet for HashMaps begyndte at bruge dedikerede klasser som Event, Match, Team, og Score. Hver klasse repræsenterer et konkret objekt med sine egne instansvariabler, hvilket giver en mere læselig måde at håndtere data.

## Import/eksport (Johannes)

Eksport og import fra/til csv-filer indebærer håndtering af de danske tegn: *æøå,* og vi havde forventet at det ville være løst ved at bruge utf-8 som format. Men det viste sig at Microsoft Excels csv-fil i utf-8 har et BOM (byte-order mark) i starten af filen (som nogle utf-8-filer har). Det nødvendiggjorde dels at slette det hexadecimale tal når vi importerer, og dels at tilføje det til den første værdi i eksportfunktionerne. Det er derfor at vi i CSVutils-funktionerne enten fjerner eller tilføjer “\uFEFF” fra/til det første element.

## Klassediagram (Martin)



Figur Klassediagram

Klassediagrammet er genereret igennem IntelliJ IDEA, med udgangspunkt i vores scr-mappe. Det viser tilhørsforholdene mellem de forskellige klasser, samt viser hvilke klasser der håndterer applikationensfunktionalitet og GUI. Diagrammet viser databasefunktionerne i DBUtils og csv-import/eksport i CSVUtils.

# Afprøvning (Johannes)

Det er klart at afprøvningen ikke kun fandt sted efter at alt var klart – i løbet af implementeringen testede vi jo hver især vores kode inden vi delte den. Men derfor brugte vi alligevel tid på systematisk at gennemprøve alle knapper og funktioner da produktet var færdigt.

Den endelige afprøvning afslørede to bugs:

**a)** Kampstart i CSV-filen manglede i kamprapporten, selvom den var tilstede i databasen. I stedet fik vi bare null i csv-filen. Det viste sig at grunden var at vores stored procedure *getGames* kun hentede de data vi viser i tabellen i GUIen, og da vi ikke henter kampstart, så indsatte DButils-funktionen blot et null som kampstart. Løsningen var at tilføje én linje til sql-koden for *getGames,* som sikrede at kampstart blev inkluderet i den hentede tabel.

**b)** Knapperne *Omdøb hold* og *Slet hold* i GUIen der viser stillingen fejlede hvis der ikke var valgt et hold i tabellen. GUIen virkede fortsat, men i java-terminalen kunne man se en kaskade af fejlmeddelelser. Løsningen var at tilføje et if-check ved begge knapper som sikrede at selektionen fra tabellen ikke var null.

Bortset fra de her to mindre bugs virkede alt faktisk som planlagt.

# Konklusion og refleksioner (Johannes og Martin)

Det endelige program kan nøjagtig de ting som blev nævnt i afsnit 3 om kravspecifikation. Ved afprøvningen stødte vi kun på to bugs, hvilket vidner om at vi var tilstrækkeligt grundige i designfasen og i implementeringen. Kort sagt, vandfaldsmodellens fire første trin (kravidentifikation, design, implementering, afprøvning) blev fulgt og med et tilfredsstillende resultat.

Vi har i rapporten inkluderet de fire diagrammer som var nødvendige ift. opgaven, og vil lade det være op til andre at vurdere om vores metodeafsnit er “lækkert”.

Vi kunne nok godt have været mere skarpe i at dele opgaverne i de tre lag: præsentation, applikation og database. Databaselaget er ret isoleret med stored procedures osv. Men som nævnt i afsnit (5.4) har vi også lagt noget af applikationslogikken i vores Controller-klasser – som styrer alt det grafiske, knapper og tabeller. Vi kunne nok også have gjort datatrafikken mellem klasser mere transparant.

En løsning kunne måske have været at benytte to greb:

1) Vi kunne have oprettet Service-klasser for hver Controller. Så kunne applikationsfunktioner ligge i Service-klasserne, og det grafiske i Controller-klasserne. Som eksempel kunne vi i vores simulation af en kamp styre det visuelle med MatchController, men oprette et Match-objekt, hente holdnavne fra databasen og beregne og gemme resultatet i en MatchService-klasse. Den kunne så have statiske funktioner ligesom vores utils-klasser, eller også kunne MatchControlleren have en MatchService-klasse som attribut. Så ville man tydeligt kunne se at metodekald som myMatchService.doStuff() hører til applikationslaget, og kald til egne funktioner (fx disableAllButtons()) hører til præsentationslaget.

2) Vi kunne lette datatrafikken ved at lave data-transfer-objects (DTO) for at sende information mellem præsentationslaget og applikationslaget. Vores Score-klasse fungerer ret godt som et DTO (og bruges direkte i CSVutils, DButils og i Controlleren), og sikrer at koden er klar og enkel de steder der overføres data mellem klasser. Den resterende datatrafik mellem klasserne burde nok ligne den. Fx så er returværdien fra CSVutils.readCSV() en liste, List<Object>, der indeholder som første element et Match-objekt, men uden ID, og det andet element er et JSONArray. Der kunne vi nok med fordel have lavet en DTO-klasse som netop havde Match og JSONArray som sine attributter med passende get-funktioner og muligvis andre nyttige funktioner.

Grundet programmets størrelse og krav, vurderer vi at vedligeholdelse ikke aktuelt. Dog er det værd at nævne at hvis programmet havde behov for vedligeholdelse, ville behovet for en større opdeling af præsentationslag og applikationslag have været en fordel.

Klassediagrammet blev først genereret efter koden var implementeret og afprøvet, så det havde ikke nogen praktisk funktion i projektet. Her ville vi gerne en anden gang være mere præcise i designfasen (jvf. en skarpere inddeling af præsentations- og applikationslag, mere klarhed i datatrafik mellem klasser) og få aftalt en struktur tydeligere end vi gjorde. Vi havde en overordnet ide om hvordan vi skulle gribe det an, men den præcise struktur opstod ud af implementeringen.

Konkret kan vi se på vores Match-klasse som eksempel på hvordan mere detaljeret klassedesign forud for implementering ville have været en fordel. Vi bruger Match til at vise kampe i kamprapport-vinduet, og vi bruger Match når vi tilføjer en ny kamp til databasen – enten gennem kampsimulationsvinduet eller gennem import fra csv-fil. Et Match-objekt har et ID, fordi det har en kamp i databasen, men faktisk har vi to slags Match – fordi en Match vi henter fra databasen, har et ID med sig, hvorimod en Match vi selv skaber (simulation eller import) bare har en defaultværdi som ID (0). Kampe får først deres kampID når de føjes til tabellen kampe (med kampID som en IDENTITY-attribut). Derfor ville det måske være bedre hvis vi havde en klasse, MatchNew, som har de relevante attributter, men ikke ID, og så en anden klasse, MatchOld, som arver fra MatchNew, men er forskellig ved at den har ID – som den jo så får fra databasen.

Fordi vi delte opgaverne ud og brugte Kanban til at distribuere dem, var der ikke overlap i vores kode, men derfor oplevede vi alligevel et par gange git-problemer. Løsningen var at oprette en git-protokol og lægge den i wikien – men det havde nok været godt at gøre det til at starte med.

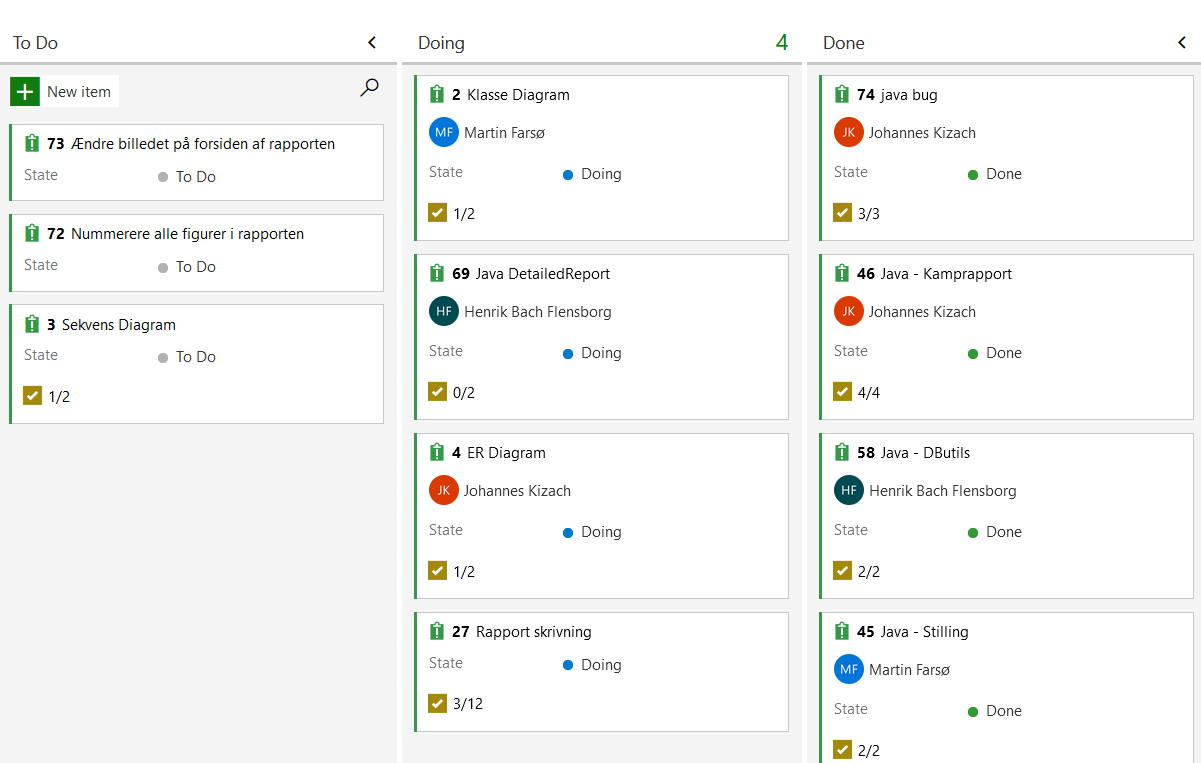
Selve brugen af DevOps har fungeret godt. Der har været en læringskurve med funktionerne og mulighederne med brugen af DevOps, men har vist sig til slut at have været et rigtig godt værktøj til at danne overblik over projektets status og tilrettelæggelse.

Vores gruppe samarbejde har været en stor succes og kontrakten har bidraget til at alle går fra projektet med nyfundet viden. Vi har bibeholdt den gode tone og skabt et godt arbejdsmiljø med gode forventninger og overkommelig arbejdsmængder, så ingen har båret projektet alene. Vi har opretholdt kontraktens indhold og sørget for at holde fast i de værdier og krav vi definerede deri, ved at gennemgå al kode og rapporten, så alle kan stå inden for hele projektet.

# Bibliografi

# Bilag

## Kanbaneksempel



## Gruppekontrakt

A document with text on it

Description automatically generated