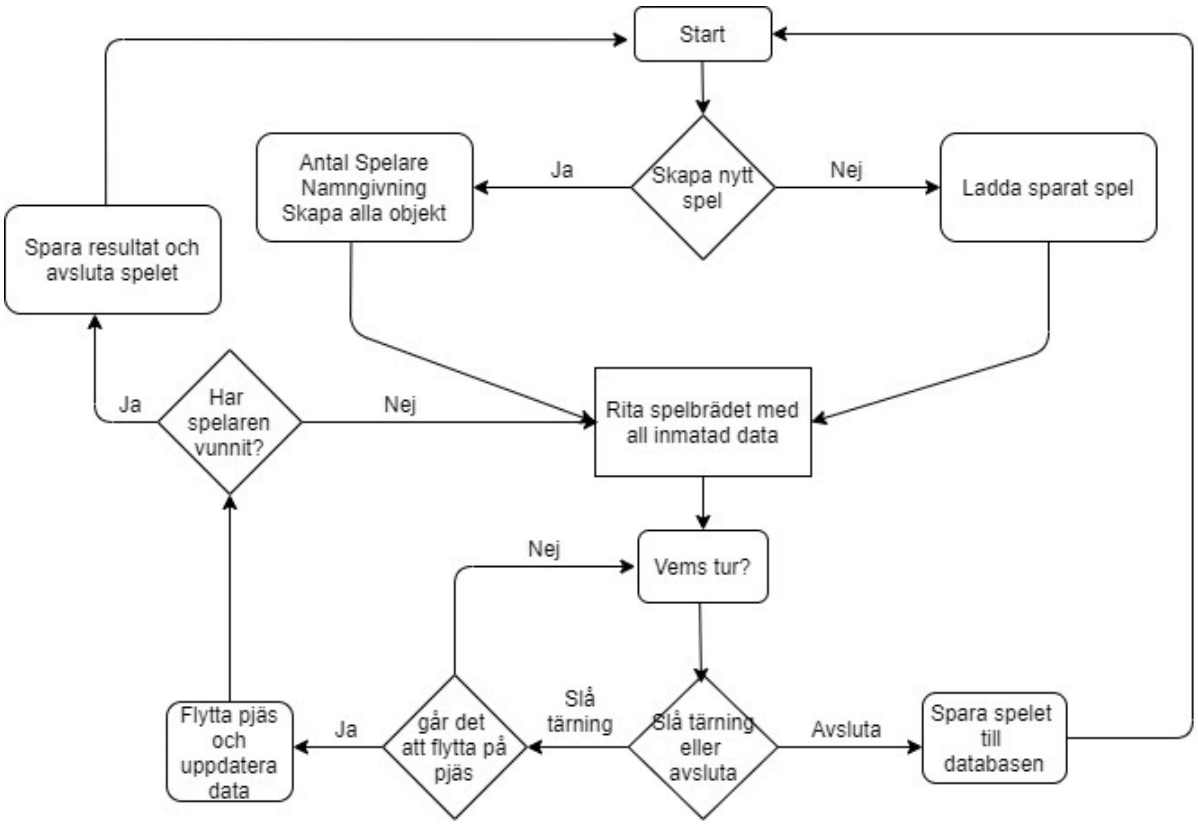
Grupp 5 - The Ludo Game

Ted Henriksson, André Morad och Sebastian Jones

User Stories:

- Som användare, ska jag ha möjlighet till att spara befintligt spel och sedan återuppta spelet vid ett senare tillfälle.
- Som användare, ska jag kunna spela med minst en till spelare.
- Som användare, ska jag kunna flytta min pjäs och vid situationer där jag har fler pjäser som går att flytta ska jag kunna välja vilken pjäs jag vill flytta.
- Som användare, vid situationer där jag har en pjäs ute på planen och slår en 1:a eller 6:a ska jag har möjlighet till att välja mellan att flytta på pjäsen ute på spelplanen eller ta ut en ny från boet.
- Som användare, ska jag ha möjlighet att kolla utgången på tidigare avslutade spel.
- Som användare, ska jag ha möjligheten att slå ut andra spelares pjäser när jag går på samma ruta.

Flowchart:



Struktur för programmet:

Regler:

- För att spelaren ska ha möjlighet till att flytta ut en pjäs från boet, behöver hen slå 1:a eller 6:a med tärningen
- Om en spelare slår en 6:a med tärningen ska spelaren få slå en gång till
- Om spelaren har fler än en pjäs på brädet får spelaren själv välja vilken pjäs hen vill flytta under sin tur.
- För att en pjäs ska gå i mål behöver spelaren slå jämt, t ex om spelarens pjäs står på en ruta från mål och slår en trea behöver spelaren gå in i mål och sedan backa resterande steg.
- Om en pjäs har gått i mål tas den ut ur spel.
- Om motståndarens pjäs kommer på samma ruta som din på spelplanen knuffas din pjäs tillbaka till boet och du behöver börja om på nytt med den pjäsen
- Spelets vinnare är den spelare som har lyckats flytta alla sina pjäser i mål först.

Klasser:

Game – Variabler

- int GameID = Primary key för tabellen i databasen
- string GameName = Namnet på spelet, användaren får själv namnge spelet för att lättare identifiera det när spelaren ska ladda spelet på nytt
- DateTime LastSaved = Uppdaterar tiden på när spelet sparades senast
- bool Finished = Anger om spelet är slut/har en vinnare
- int PlayerTurn = Fungerar som en indexering till Players listan, den börjar på 0 och adderas med varje gång nuvarande spelaren har spelar sin runda. Om PlayerTurn skulle bli högre än vad det finns spelare nollställs den och börjar om på nytt.
- int Round = Räknar hur många rundor som spelat har pågått, den adderas med ett varje gång när PlayerTurn nollställs.
- List<Player> Players = Kopplar ihop Player-tabellen i databasen.

Metoder:

- **CreatePlayer**() = Skapar spelare och ger fyra tokens till spelaren
- **UpdateTurnAndRound**()= Uppdaterar vems tur det är samt vilken runda det är.
- **CheckWinner**() = Kolla om någon spelare har vunnit, om en spelares alla pjäser i mål sätts Finished varibalen till true.
- **NameTheGame**() = sätter GameName varibalen.
- **GetVictoriousPlayer**() = hämtar ut den segrande spelaren från Player listan i spel objektet.

Player – Variabler

- int PlayerID = Primary key för tabellen i databasen
- Game Game = För att koppla ihop spelar objektet till Gametabellen
- string PlayerName = Namnger spelaren
- string PlayerColor = Anger vilken färg spelaren har
- bool Winner = Anger om spelaren har vunnit eller inte
- List<Token> Tokens = Kopplar ihop Token-tabellen i databasen

Token – Variabler

- int TokenID = Sätter Primary key i Token-tabellen
- Player Player = Kopplar ihop Token till Player-tabellen
- string TokenColor = Anger vilken färg pjäsen har
- int TokenNumber = Anger vilket nummer pjäsen har, som vi sätter 1-4 på spelarens pjäser
- int GameBoardPosition = Sätter pjäsens position på yttre brädet/vita rutorna
- int StepsCounter = Räknar hur många steg pjäsen har tagit, varje pjäs ska gå 1-45 för att gå imål
- bool InNest = För att ange om pjäsen är i boet, sätts som true i början eller när pjäsen blir utknuffad och sätts som false när pjäsen lämnar boet
- bool InGoal = Sätts som true när pjäsen har gått 45 steg
- bool InEndLap = Sätt som true när pjäsen har gått fler än 40 steg, när InEndLap sätts som true är den immun från att bli utknuffad.

Metoder:

- **CountTokenSteps**(Token currentToken, int dieResult) = Uppdaterar pjäsens **StepCounter** med utfallet på tärningskastet
- **AtEndLap**() = kontrollerar om pjäsen är på “upploppet”, om pjäsens StepCounter är över 40 sätts pjäsens InEndLap till true.
- **CountGameBordPosition**(int dieResult) = Uppdaterar pjäsens GameBoardPosition med hjälp av tärningen.
- **TokenInGoal**() = Kontrollerar om pjäsen har kommit i mål, om pjäsens StepCounter == 45 ska pjäsens InGoal sättas till true.
- **TokenStartGameBoardPosition**(Token setStart) = Sätter pjäsens “startposition” på brädet, eller rättare sagt från vilket värde den ska börja räkna ifrån.

Engine

ThrowDie() = Simulerar ett tärningskast och returnerar en int variabel mellan 1-6.

MovableTokens(Player currentPlayer, int dieResult) = Returnerar en lista med pjäser som spelaren kan flytta på, beroende på utfallet av tärningen och alla pjäsers positioner på brädet. Den valda pjäsen kommer sedan att skickas in i ChooseToken.

ChooseToke(List<Token> tokensToPlay, Token tokenToMove) = Tar fram den pjäs som spelaren vill flytta på för att senare skickas in i RunPlayerTurn.

RunPlayerTurn(Token currentToken, int die, Game game, Player currentPlayer)
Denna metod sköter mer eller mindre alla data uppdateringar som sker under spelarens tur när hen vill flytta på en pjäs. Metoden kör dessa steg:

- Ändra valdToken.InNest = false, om pjäsen flyttas ut ur nästet.
- Uppdaterar pjäsens StepCounter i metoden CountTokenSteps med resultatet från die
- Kolla om pjäsen befinner sig på "upploppet" via metoden AtEndLap, om pjäsens steg är mer än 40 sätts InEndLap till true, annars false.
- Uppdaterar pjäsens GameBoardPosition i metoden CountBoardGamePosition med hjälp av tärningens utfall(die).
- Kontrollerar om det finns en pjäs från en motståndare på samma GameBoardPosition via metoden KockOutAnotherToken.
- Kollar om pjäsen har gått imål via metoden TokenInGoal
- Kollar om spelaren har vunnit via metoden CheckForWinner
- Kontrollerar om spelaren får slå en gång till via if-satsen, om die inte har värdet 6 samt om spelet inte har en vinnare. Om dessa kravs uppfylls uppdateras spelet om vems tur det är.

RunGameUpdate(Game game)

Denna metod körs enbart om en spelare inte kan flytta några pjäser, metoden uppdaterar Game objektets playerturn samt vilken runda det är.

SaveGameToDataBase(Game game)

Metoden används för att spara spelet till database, här kör vi en try catch för att metoden ska första testa att uppdatera spelobjektets data i databasen om spelobjektet redan finns där. Om spelobjekt inte finns i databasen ska metoden köra catch för att spara objektet till databasen.

LoadPreviousGamesFromDataBase()

Metoden returnerar en lista på alla ofärdiga spelobjekt, alltså alla spelobjekt som har sin Finished variabel som false. Vi använder oss utav Include för att hämta alla Player objekt som tillhör spelobjektet och sedan ThenInclude för att hämta alla Token objekt som tillhör spelarna. Denna metod används för att ge användaren en möjlighet till att återuppta ett tidigare spel.

LoadAllFinishedGamesFromDataBase()

Metoden returnerar en lista på alla spelobjekt som har spelats färdigt, den är uppbyggd på mer eller mindre samma sätt som *LoadPreviousGamesFromDataBase* metoden. Det som skiljer är att denna metod ska returnera alla spelobjekt som har variabeln Finished som true.

KnockOutAnotherToken(Token currentToken, Game game)

Denna metod används för att kolla om två pjäser från två olika spelare hamnar på samma GameBoardPosition, så ska currentToken knuffa ut den pjäs som redan står på samma GameBoardPosition.

Linq-metoden vi använder är att den hämtar ut alla pjäser som är med i spelet, förutom den spelare som kör sin tur. Som exempel om vi har tre spelare i spelet, spelare röd, spelare blå och spelare grön. Just nu är det "röda spelaren" som kör sin tur och då tar linq-metoden enbart fram de blåa och de gröna pjäserna för att matchas mot den röda pjäsens GameBoardPosition.

Om den röda pjäsen får en träff mot någon utav de blåa eller gröna pjäserna, så sätts den pjäs alla värden till startläget.

LudoContext

I databasen använder vi oss utav tre tabeller, Games, Players och Tokens, deras relationer till varandra är att Games-tabellen har en One-To-Many relation till Players-tabellen samt att Player-Tabellen har en One-To-Many relation till Tokens-tabellen.

Vi valde att använde oss utav OnModelCreating för att sätta alla primary keys på tabellerna samt vid skapandet av tabellernas relationer till varandra, men även till att ändra på kolumn typen för LastSaved i Games-tabellen till *smalldatetime*. Vi använde oss utav en DateTime variabel för LastSaved i själva klassen vilket gjorde att EnitytFramwork översatte denna variabel till datetime2 som kolumntyp i vår SQL-databas, vilket i sin tur gjorde att den lagrade tusendelar utav en sekund. Att visa den exakta tusendel av en sekund var väldigt onödigt i vår mening och därför tog vi beslutet att ändra på kolumnen till en *smalldatetime* för att det skall bli mer lättläst men även för att ta bort onödig tidsinformation.

Vi testade att köra denna metod till att DateTime skulle uppdateras utav sig självt varje gång när spelet sparas till databasen:

```
modelBuilder.Entity<Game>().Property(g => g.LastSaved).
HasColumnType("SMALLDATETIME").
HasDefaultValueSql("SYSDATETIME()").
ValueGeneratedOnAddOrUpdate();
```

Metoden fungerade för att uppdatera LastSaved kolumnen vid varje tillfälle när spelet sparas, dock uppdaterade den åtta timmar innan vår tid. Vi antog att *SysDateTime* uppdaterar efter amerikansk tid direkt eller så är det på grund utav att vi använder oss utav GearHost för att lagra data och att *SysDateTIme* uppdaterar efter den lokala tiden hos den servern som vår databas ligger på. Där av tog vi beslutet att ta bort denna metod och istället låta spelmotorn uppdatera LastSaved variabeln direkt innan spelet sparas till databasen, för att få det "lite snyggare".