

Projektuppgift

DT071G, Programmering i C# .NET

Matchklocka för ishockey

Fredrik Eklund



Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. Campus Sundsvall Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MITTUNIVERSITETET
Avdelningen för informationssystem och -teknologi

Författare: Fredrik Eklund, frek1802@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin, år: 02, 2024

Sammanfattning

Syftet med arbetet har varit att skapa en matchklocka för ishockey som är utformad för att uppnå kraven för en godkänd matchklocka enligt Svenska Ishockeyförbundets kravställning. Applikationen är skapad i språket C# och ramverket .NET.

Innehållsförteckning

Sammanfattning.....	iii
Innehållsförteckning.....	iv
Terminologi.....	1
1 Introduktion	2
1.1 Bakgrund och problemmotivering.....	2
1.2 Konkreta och verifierbara mål	2
1.2.1 Kriterier för slutfört projekt:	2
1.3 Avgränsningar	2
2 Teori	4
2.1 C#.....	4
2.2 .NET	4
2.3 UDP	4
2.4 Async och await i C#/.NET	4
2.5 Tasks och Threads i C#/.NET	4
2.6 Event i C#/.NET	5
3 Metod.....	6
3.1 Utvecklingsmiljö.....	6
3.2 Systemdesign och planering	6
3.3 Versionshantering	7
3.4 Tekniska överväganden	7
3.4.1 UDP som protokoll för nätverkskommunikation.....	7
3.4.2 Strategi för återställning av match	7
3.5 Test av applikation	7
4 Konstruktion.....	9
4.1 Planering och research.....	9
4.1.1 Flödesschema / diagram	9
4.1.2 Systemdesign	9
4.1.3 Designskiss kontroll.....	9
4.2 Utveckling av matchklockans logik.....	10
4.2.1 Matchklocka / timer – BaseTimer	10
4.2.2 GameClock – Styr aktiv klocka/timer	11
4.2.3 GameSettings	12
4.2.4 Game – Länkar alla instanser i matchklockan	13
4.2.5 GameScore – poängställningen i matchen	15

4.2.6	GamePeriod – Håller koll på perioder.....	15
4.2.7	GamePenalties – Hanterar utvisningar.....	16
4.2.8	Penalty	17
4.3	Utveckling av kontroll och display för matchklocka.....	17
4.3.1	ConsoleDisplay – Visar den aktuella informationen.....	18
4.3.2	GameController – Kontrollera matchklockan.....	18
4.4	Övrig funktionalitet	20
4.4.1	GameJsonSerialzier	20
4.4.2	FileManager.....	20
4.4.3	GameStateLoader.....	20
4.4.4	StartupDialog	20
4.4.5	GameSettingsManager	21
4.4.6	UdpTransmitter	21
4.5	Test av applikation	21
5	Resultat.....	23
5.1	Kriterier för slutfört projekt:	23
6	Slutsatser	25
7	Referenser	27
8	Bilagor	29
8.1	Bilaga A – Diagram och flödesscheman	29
8.1.1	Diagram över applikation.....	29
8.1.2	Flödesschema för att beräkna tid	30
8.1.3	Flödesschema för att lägga till en ny utvisning.....	31
8.2	Bilaga B – Skärmdump från konsolen.....	32
8.3	Bilaga C.....	33

Terminologi

Akronymer/Förkortningar

JSON	JavaScript Object Notation. Är en vanlig standard för att formatera data som ska skickas eller lagras.
C#	Är ett objektorienterat programmeringsspråk.
.NET	Är en plattform/ramverk för att utveckla plattformsoberoende applikationer.
IDE	Integrated Development Environment.
UDP	User Datagram Protocol
API	Application Programming Interface
SIF	Svenska Ishockeyförbundet

1 Introduktion

1.1 Bakgrund och problemmotivering

Idag finns det flera matchklockor som är byggd på gammal hårdvara, eller är alldeles för dyra i inköp. Detta gör det svårt för föreningar att ha råd att köpa ny utrustning när något går sönder, eller ha möjligheten att uppgradera till en modernare lösning.

1.2 Konkreta och verifierbara mål

Projektets mål är att i så stor utsträckning som möjligt bygga en applikation som uppfyller Svenska Ishockeyförbundets (SIF) krav för en godkänd matchklocka, som är listade i "Regelbok Anläggningar" (1). Applikationen ska kunna köras på befintlig hårdvara och vara plattformsoberoende.

1.2.1 Kriterier för slutfört projekt:

- Hantera matchtid räknad uppåt alternativt neråt. Starta, stoppa och justera tid, samt stöd för att hantera övertidsperioder med annan periodlängd. Om tiden är mindre än 1 minut ska tiden visas i sekunder och tiondelar, annars i minuter och sekunder.
- Hantera utvisningar. Funktionalitet för att lägga till, ta bort, ändra och visa utvisningar.
- Hantera mål. Funktionalitet för att lägga till, ta bort och visa.
- Funktionalitet för Time-out och powerbreaks, räknad nedåt till 0 sekunder.
- Funktionalitet för att hantera paustid, räknad nedåt från det totala antalet minuter till 0.
- Funktionalitet för att kunna återställa tidigare match efter strömavbrott eller vid händelse att applikationen kraschar.
- Möjlighet att skicka klockans status/signal över nätverket till displayer/skrämar, klockor i omklädning/domarrum och eventuell tv-produktion via UDP Multicast.

1.3 Avgränsningar

Det här projektet är avgränsat till att skapa en konsolapplikation med C# och .NET. Huvudfokus ligger på att utveckla logiken och funktionaliteten för matchklockan. Inom ramen för detta projekt kommer det inte att byggas ett grafiskt gränssnitt för att kontrollera matchklockan, men tanken är att applikationen ska struktureras på ett sätt att det inte krävs omfattande justering av koden för att bygga på ett grafiskt gränssnitt.

Projektet innefattar inte heller utvecklingen av en visuell resultattavla som visar informationen för publiken, utan fokuserar på utvecklingen av matchklockans funktionalitet.

2 Teori

2.1 C#

C# är ett modernt, objekt-orienterat programmeringsspråk med öppen källkod, utvecklat av Microsoft. Det är det vanligaste språket för att utveckla applikationer med plattformen/ramverket .NET. (2)

2.2 .NET

.NET är en plattform eller ett ramverk för att utveckla driftsäkra, plattformsoberoende applikationer med hög prestanda. Det innehåller en hel del bibliotek som täcker det mesta för att utveckla applikationer, som till exempel nätverk och maskininlärning, och om inte standardbiblioteken räcker till finns det över 300 000 paket att hämta via .NET pakethanterar "NuGet". (3)

2.3 UDP

User Datagram Protocol (UDP) är ett protokoll som används för att skicka data över nätverket. Protokollet är mycket snabbt, men är inte 100% pålitligt, det tar nämligen inte hänsyn till om något paket som skickas inte når fram, eller kommer i fel ordning, men enkelhet och snabbheten i protokollet gör det till en bra val där inte paketförluster spelar lika stor roll. Det är vanligt att det används inom datorspel samt IP-telefoni då dessa användningsområden kan vara något mer förlåtande för paketförluster. (4)

2.4 Async och await i C#/.NET

C# och .NET har bra stöd för att skriva asynkron kod med hjälp av "async", "await".

Async används för att markera att en metod eller en funktion är asynkron, och en sådan metod/funktion bör innehålla minst ett anrop till en metod/funktion med hjälp av "await".

Await fungerar som så att den tillfälligt hindrar en asynkron metod/funktion från att gå vidare, den måste vänta in ett svar från den anropade metoden/funktionen innan den får gå vidare. Ett vanligt användningsområde för "await" är när man gör ett anrop till ett Application Programming Interface (API) för att till exempel hämta data från en databas eller annan extern tjänst, då används "await" för att man ska invänta svaret från ett API innan man går vidare och försöker behandla data från anropet. (5)

2.5 Tasks och Threads i C#/.NET

En "Task" är ett objekt som representerar ett asynkront arbete som ska utföras i en applikation. En "Task" körs vanligtvis i en "Thread" som representerar en tråd i datorns processor.

Dessa "Threads" hanteras av en "ThreadPool" som tilldelar en "Task" en "Thread", eller skapar nya vid behov. Behovet av "Threads" kommer när

man behöver utföra flera olika operationer inom en applikation utan att låsa huvudtråden. (6)

2.6 Event i C#/.NET

Ett event i C# är ett sätt för en klass att meddela till andra klasser att något har hänt. En klass kan prenumerera på uppdateringar från ett event i en annan klass, och kan då anropa metoder i sin tur. I .NET finns möjligheten att skicka med data eller argument med eventet som en prenumerant kan ta emot. Detta görs genom att använda "EventHandler-biblioteket" som kan skicka med "EventArgs", och en prenumerant kan då ta emot och behandla denna data. (7) (8).

3 Metod

3.1 Utvecklingsmiljö

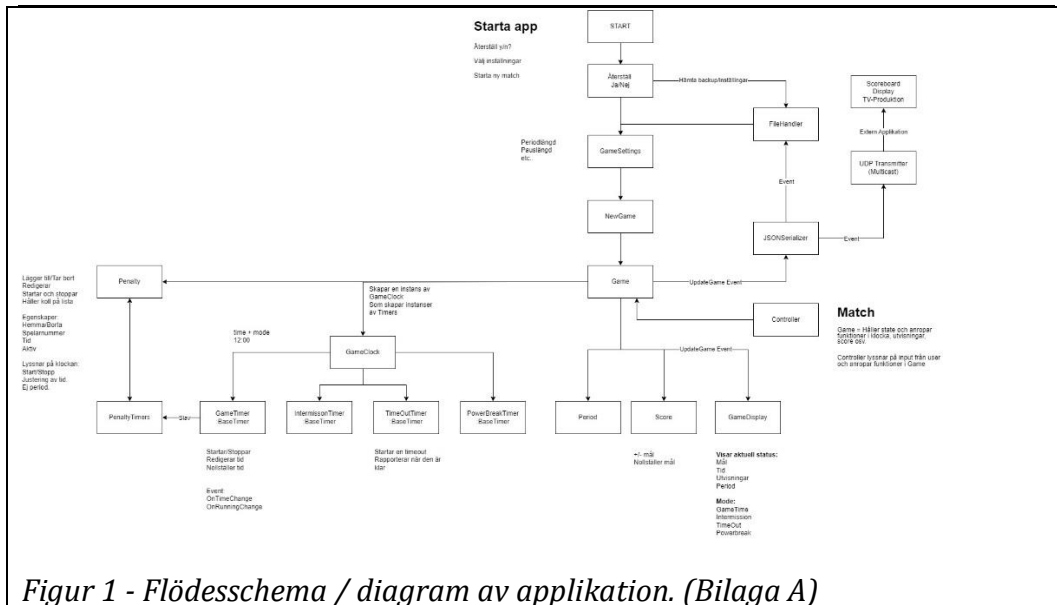
Visual Studio Code / Visual Studio 2022	Kodeditor / Integrated Development Environment (IDE)
Windows 11, Pop!_OS 24.04	Operativsystem
GitHub	För versionshantering
Powershell	Kommandotolk, använd för att använda git och .NET kommandon
Draw.io	Flödeschema/diagram
Figma	Används för att skapa designskiss för applikationens grafiska gränssnitt
Wireshark	Används för att analysera och kontrollera nätverksanrop. (9)

3.2 Systemdesign och planering

Applikationen är inte utvecklad utefter något designmönster, men grundtanken är att varje klass ska vara så smal som möjligt, och informationen ska flöda genom applikation med hjälp av event.

Klassen för att hantera mål ska till exempel enbart hantera mål. Den ska innehålla egenskaper och metoder för att hålla koll och hantera poängställningen i matchen. Detta gör att det blir enklare att lägga till och ta bort moduler om man i framtiden skulle behöva ha stöd för fler sporter än ishockey.

En planering för de olika klasserna och hur dessa ska kommunicera har ritats upp i ett flödesschema eller diagram som kan ses i figur 1 nedan eller i **Bilaga A**. Detta används som stöd för att bygga applikationen.



Figur 1 - Flödesschema / diagram av applikation. (Bilaga A)

3.3 Versionshantering

Under utvecklingen av applikationen kommer versionshantering av koden att skötas med git och GitHub. Utveckling kommer att ske i en utvecklingsbranch, för att man inte ska riskera att förstöra eller ändra något i main-branchen.

3.4 Tekniska överväganden

3.4.1 UDP som protokoll för nätverkskommunikation

För projektet valdes UDP som protokoll för nätverkskommunikationen främst för att det föreslogs i SIF:s "Regelbok Anläggningar" att matchklockans signal ska skickas över nätverket med Multicast vilket med fördel görs med UDP då detta möjliggör att information skickas i realtid till flera enheter samtidigt utan att det blir en belastning för servern/enheten som skickar data. (1) (10)

3.4.2 Strategi för återställning av match

För att ha möjligheten att återställa en match efter en oförutsedd händelse uppstått skrivs information till en temporär JSON-fil varje gång någon ändring i matchens status sker, som till exempel att tiden tickar eller ett mål sker.

Efter att ha lyckats med operationen att skriva data till filen, skrivs sedan en annan JSON-fil över, vilket är den som kan användas för att återställa matchens status. Denna fil laddas in när applikationen startar och en användare kan då välja att återställa en match.

3.5 Test av applikation

Applikationen testas genom att testa hur matchklockan hanterar vanliga scenarier som uppstår under en match, som till exempel att matchtiden ska justeras eller att en utvisning ska tas bort innan utvisningstiden löpt ut. Den

testas även på Windows och Linux för att säkerställa att den kan köras på olika plattformar.

4 Konstruktion

4.1 Planering och research

Först och främst gjordes en övergripande planering av funktioner som en matchklocka bör innehålla för att vara fungerande. Utifrån SIF:s "Regelbok Anläggningar", en manual från "Westerstrands sportsystem" för en enklare matchklocka, samt min egen kunskap om ishockey skapades en enklare kravlista som låg till grund för flödesschemat av applikationen. (11) (1)

4.1.1 Flödesschema / diagram

Utifrån den inhämtade och summerade informationen, skapades ett enklare diagram för att visa olika klasser, samt hur informationen flödade genom systemet. Diagrammet har justerats under utvecklingens gång, då bland annat matchklockan bröts ut i flera olika klockor, till exempel paus- och timeout-klockor.

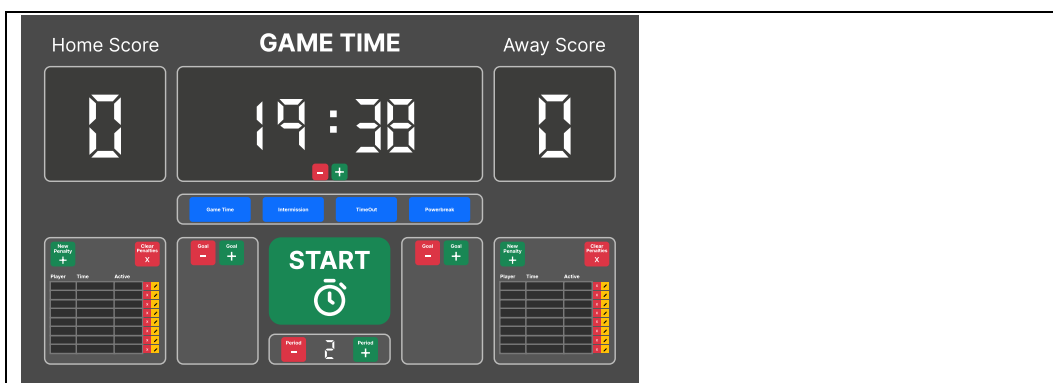
4.1.2 Systemdesign

Målet med applikationen är att ha en fungerande matchklocka, som består av en huvudklass som skapar instanser till övriga klasser/moduler samt ansvarar för att informationen flödar igenom applikationen. Målet är att bygga flera fristående klasser/moduler som ska kunna läggas till eller tas bort, för att i framtiden kunna anpassa matchklockans funktionalitet efter behov och olika sporter.

Som tidigare nämnt är inte applikationen utvecklad med något designmönster i åtanke, men det finns viss inspiration från SOLID-principen, med att målet att en klass enbart ska ansvara för en sak, till exempel att hantera mål/poängställning. Detta för att göra det så enkelt som möjligt att göra applikationen så flexibel och modulär som möjligt.

4.1.3 Designskiss kontroll

Efter att applikationen var skapad, fanns en förhoppning att ett grafiskt gränssnitt för att kontrollera matchklockan skulle hinna implementeras, men tid fanns inte till mer än en enklare designskiss/wireframe som kan ses i figur 2 nedan.



Figur 2 - Designskiss kontroll för matchklocka

4.2 Utveckling av matchklockans logik

4.2.1 Matchklocka / timer – BaseTimer

Metoder i BaseTimer:

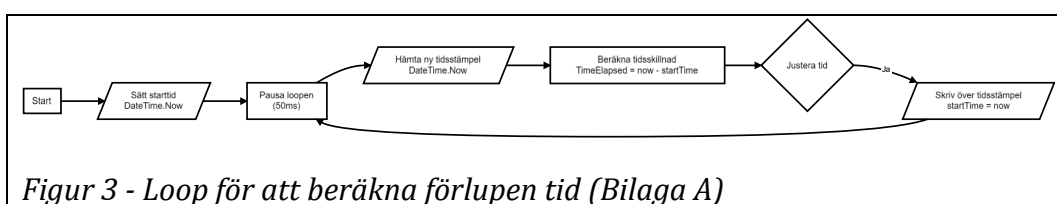
- **StartClockAsync** – Startar en task som kör en while-loop som anropar övriga metoder i klassen. Anropar metoder som triggar eventet "OnTimerUpdated" och "OnTimerEnded".
- **Stop** – Stoppas loopen
- **UpdateTime** – Räknar ut förlupen matchtid och justerar matchtiden/klockan.
- **AdjustTime** – Används för att kunna justera klockan framåt eller bakåt
- **SetCurrentTime** – Används för att sätta klockan till en specifik tidpunkt.
- **ShouldStop** - Används för att bestämma om en klocka ska stoppas för att tiden löpt ut.

Klasserna GameTimer, TimeoutTimer, IntermissionTimer och PowerbreakTimer ärver från BaseTimer.

Det första som utvecklades var möjligheten att starta och stoppa matchtiden. Det här var den viktigaste delen och egentligen kärnan i hela applikationen, för om man inte kan mäta tid, eller kontrollera tiden så finns det ju egentligen ingen vits med en matchklocka.

Den första implementationen använde sig av den inbyggda klassen "Stopwatch", då denna har möjligheten att mäta tid på ett precist sätt. Men den implementationen valdes bort då jag själv vill ha mer kontroll på hur tiden mäts, och jag landade till slut i att det bästa sättet var att använda sig av tidstämplar och räkna ut förlupen tid mellan dessa tidsstämplar. Den aktuella tiden hålls i en TimeSpan. (12) (13)

Detta görs genom att metoden StartClockAsync, startar en loop som anropar metoden UpdateTime, som tar tidstämplar och beräknar förlupen tid. Hur loopen går till kan ses i figur 3 nedan och Bilaga A.



Figur 3 - Loop för att beräkna förlupen tid (Bilaga A)

4.2.1.1 *GameTimer: BaseTimer*



Figur 4 - Metoden *UpdateTime* i klassen *BaseTimer*.

Det som skiljer *GameTimer* från de övriga klasserna är att den gör en "override" på *AdjustTime*-metoden, och *UpdateTime*-metoden som kan ses i figur 4 ovan. Anledningen är att *GameTime* klassen ansvarar för att uppdatera utvisningstiden för de aktiva utvisningarna.

4.2.2 **GameClock – Styr aktiv klocka/timer**

För att hantera de olika klockor/timers skapades en klass som heter *GameClock*. Den här klassen ansvarar för att skapa instanser av de olika typerna av klockorna och anropa dess metoder. I ishockey visas alltid en aktiv tid på en resultattavla, vilken kan vara matchtid, paustid, timeout-tid med flera. Egenskapen "ActiveTimer" håller reda på vilken klocka/timer som är aktiv, och kan anropa dess metoder. Alla timers har olika längder, dessa styrs av *GameSettings* som väljs när applikationen startar.

Klassen innehåller följande metoder:

- **Activate- :**
 - **TimeOut** – Aktiverar timeout-läge. Skapar en ny instans av *TimeOutTimer* och sätter denna instans som aktiv timer.
 - **Intermission** – Aktiverar paus-läge. Skapar en ny instans av *IntermissionTimer* och sätter denna instans som aktiv timer.
 - **GameTime** – Sätter en tidigare skapad instans av *GameTimer* som aktiv timer.

- **Powerbreak** – Aktiverar powerbreak-läge. Skapar en ny instans av PowerbreakTimer och sätter denna instans som aktiv timer.
- **NewPeriodTimer** – Skapar en ny instans av GameTimer, vilket kan likställas med att skapa en ny period.
- **AdjustTime** – Anropar metoden AdjustTime för den aktiva timern.
- **SetCurrentTime** – Anropar metoden SetCurrentTime för den aktiva timern.
- **StartActiveClock** – Anropar metoden StartClockAsync för den aktiva timern.
- **StopActiveClock** – Anropar metoden Stop för den aktiva timern.

Klassen har även tre publika egenskaper som är viktiga för klassen, det är GameTimer, som är en instans av klassen GameTimer, detta gör att det alltid går att hämta information om matchtid. En privat egenskap _gameSettings som innehåller inställningar för matchen. Samt egenskapen ActiveTimer som är instansen mot den aktiva klockan, och det är genom den som matchklockan styrs. I figur 5 nedan kan man se ett exempel på hur ActiveTimer används för att starta klockan.

```
public void StartActiveClock()
{
    _ = ActiveTimer.StartClockAsync();
}
```

Figur 5 - Metod för att starta den aktiva klockan/timern med ActiveTimer

4.2.3 GameSettings

Vid uppstart av applikationen kan man välja inställningar för matchen. Detta är nödvändigt för att till exempel seniormatcher och ungdomsmatcher har bland annat olika periodlängd och pauslängd samt att klockan brukar räkna uppåt i ungdomsmatcher, och nedåt i junior- och seniormatcher. (14)

```
public TimeSpan PeriodLength { get; private set; }
public TimeSpan OvertimePeriodLength {get; private set;}
public TimeSpan TimeOutLength {get; private set;}
public TimeSpan PowerbreakLength {get; private set;}
public TimeSpan IntermissionLength {get; private set;}
public int NumberOfPeriods { get; private set; }
public bool Countdown {get; private set;}
```

Figur 6 - Egenskaper i klassen GameSettings

När en ny instans av GameSettings skapas efter anrop i Program.cs sätts flera olika inställningar gällande matchen, som kan ses i figur 6 ovan.

4.2.4 Game – Länkar alla instanser i matchklockan

Den här klassen är "spindeln i nätet" för att länka ihop applikationens olika delar, och innehåller eventet "UpdateGame" som ansvarar för att sprida information om matchstatusen runt i applikationen. Den ansvarar för att skapa instanser av alla klasser som används för att hålla koll på och styra matchens status, som matchklocka, poängställning, utvisningar och så vidare. Genom dessa publika instanser kan andra klasser som har en instans av Game-klassen, anropa metoderna i dessa instanser och på så vis styra matchen. Det gör det möjligt att enkelt byta ut en kontrollenhet eller bygga ett API för att styra matchklockan i framtiden. Det är även här logiken för att återställa en tidigare match finns.

```
public Game(GameSettings settings, bool isRestore, GameEventArgs? restoreData)
{
    Settings = settings;
    GameScore = new();
    GamePeriod = new(settings);
    GamePenalties = new GamePenalties();
    GameClock = new GameClock(settings, GamePenalties);
    ConsoleDisplay = new ConsoleDisplay(this);

    // If isRestore is true, call method to restore game
    if (isRestore && restoreData != null)
    {
        RestoreGameState(restoreData);
    }

    // Listen for events
    RegisterForUpdates();
    GameScore.ScoreChanged += OnGameChanged;
    GamePeriod.PeriodChanged += OnGameChanged;
    GamePenalties.PenaltyChanged += OnGameChanged;
}
```

Figur 7 - Konstruktör för klassen Game

I Program.cs skapas en instans av klassen, där den får med en instans av klassen GameSettings som sedan används för att kunna skapa upp timers med korrekt längd och inställningar. Även ett booleskt värde skickas med samt ett objekt innehållande den tidigare matchens status för att ha möjlighet att återställa. Konstruktorn för klassen syns i figur 7 ovan.

De viktigaste metoderna i klassen Game:

- **RegisterForUpdates** – Den här metoden registrerar sig för att lyssna på den aktiva timern/klockan i klassen GameClock
- **UnRegisterForUpdates** – Den här metoden avregistrera sig för att lyssna på den aktiva timern/klockan i klassen. Används när den aktiva klockan växlar, till exempel när klockan växlar mellan matchtid och timeout.
- **OnTimerUpdated, OnGameChanged** – Dessa två metoder triggas av event från andra klasser, till exempel om poängställningen triggas, klockan tickar eller en utvisning läggs till. Triggerar i sin tur metoden Update.
- **Update** – Skapar en ny instans av klassen GameEventArgs, och triggerar eventet UpdateGame. Det är genom den här metoden och eventet som all info om matchens status sprids.
- **RestoreGameState** – Är en metod som används för att återställa en tidigare matchs status och kan ses i figur 8 nedan.

```
private void RestoreGameState(GameEventArgs restoreData)
{
    GameScore.SetScore(restoreData.HomeScore, restoreData.AwayScore);
    GamePeriod.SetCurrentPeriod(restoreData.CurrentPeriod);
    GameClock.SetCurrentTime(restoreData.GameClockCurrentTime);
    GamePenalties.RestorePenaltyLists(restoreData.HomePenalties, restoreData.AwayPenalties);
}
```

Figur 8 - Metoden RestoreGameState används för att återställa en tidigare matchs status

Förutom dessa metoder finns ett antal metoder som jag valt att kalla för "macros", dessa anropar metoder i flera olika instanser för att till exempel växla till nästa period. Anledningen till att dessa skapats är för att spara tid vid utveckling av ny kontrollenhet för att styra matchklockan. Då slipper man skriva mer komplexa metoder för att till exempel byta period. Ett exempel på metoden NextPeriod kan ses i figur 9 nedan.

```
public void NextPeriod()
{
    GameClock.StopActiveClock();
    UnregisterFromUpdates();
    GamePeriod.IncrementPeriod();
    var periodLength = GamePeriod.GetPeriodLength();
    GameClock.NewPeriodTimer(periodLength, GamePenalties, Settings.CountDown);

    RegisterForUpdates();
    Update();
}
```

Figur 9 - Metoden NextPeriod, ett macro för att byta period

4.2.4.1 *GameEventArgs*

Den här klassen används för att paketera informationen som ska skickas vidare i applikationen, framför allt genom eventet `UpdateGame`. En instans av klassen innehåller all information som finns om matchen, som matchtid, utvisningar, perioder, matchinställningar, och poängställningar. Den här information är den som skrivs till JSON-filen för att kunna användas som återställningsfil, men också den data som skickas via UDP till bland annat en resultattavla.

De viktigaste egenskaperna i klassen:

- **CurrentTime** - Innehåller den aktiva klockan/timerns aktuella tid, används för att kunna visa aktuell tid på en resultattavla.
- **GameClockCurrentTime** - Innehåller `GameTimers` aktuella tid, den är viktig för att kunna återställa matchens tid om applikationen skulle krascha under en timeout eller powerbreak.
- **Home-/ AwayScore** - Innehåller poängställning för de båda lagen.
- **Home-/ AwayPenalties** - Innehåller en lista med respektive lags utvisningar.
- **GameSettings** - Innehåller matchens inställningar, används när matchen ska återställas.

4.2.5 *GameScore* – poängställningen i matchen

Det här är en enkel klass, den innehåller egenskaper som håller koll på poängställningen i matchen för respektive lag.

Det finns tre metoder som är viktiga i klassen:

- **AddGoal** - Läger till mål för ett lag. Tar in en "int" som styr vilket lag målet tillhör. 0 för hemmalag och 1 för bortalag. Trigger metoden `UpdateScore`.
- **RemoveGoal** - Tar bort ett mål för ett lag, använder sig också av en "int" för att bestämma lag. Den här metoden tar bort ett mål, men använder sig av "Math.Max" för att förhindra negativa värden. Trigger metoden `UpdateScore`. (15)
- **UpdateScore** - Trigger eventet "ScoreChanged" som lyssnas på i `Game`-klassen, och sin tur trigger eventet "UpdateGame". Detta säkerställer att poängställning alltid är uppdaterad.

4.2.6 *GamePeriod* – Håller koll på perioder

En enkel klass vars uppgift är att hålla koll på den aktiva perioden, och innehåller logik för att avgöra om det är en övertidsperiod. Den är väldigt lik `GameScore`-klassen, då den har metoder för att öka och minska perioder och har egenskaper för att hålla koll på aktiv period och ett booleskt värde för att hålla koll på om det är övertid eller inte. Den har också egenskaper som

totala antal perioder (iny), periodlängd för vanliga och övertidsperioder (TimeSpan), den informationen sätts i konstruktorn med hjälp av instansen för GameSettings.

De viktigaste metoderna:

- **Increment-/DecrementPeriod** – Justerar aktiv period. Och anropar metoden "CheckIfOvertime".
- **CheckIfOvertime** – Kollar om den aktiva perioden är större än "NumberOfPeriods", sätter i sådant fall "IsOvertime" till true och triggerar även metoden Update som är en metod som triggerar eventet "PeriodChanged" Som lyssnas på i Game-klassen.
- **GetPeriodLength** – En metod som returnerar en TimeSpan med periodlängden, baserat på om det är en övertidsperiod eller inte. Exempel på anrop till metoden finns i figur 9 ovan.

4.2.7 GamePenalties – Hanterar utvisningar

Den här klassen ansvarar för att hantera alla utvisningar som sker under match. Den innehåller metoder som håller koll, lägger till, ändrar, tar bort och håller koll på aktiva utvisningar. Varje utvisning är en instans av klassen Penalty.

De viktigaste egenskaperna är:

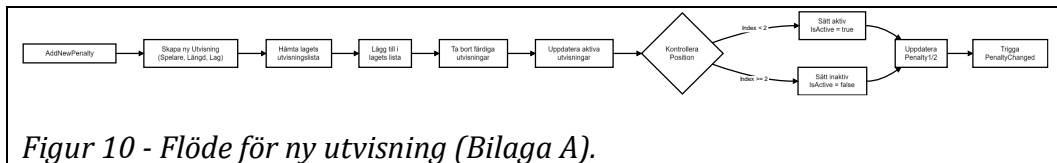
- **Home-/AwayPenalty** 1 och 2 – Den här egenskapen representerar de aktiva utvisningar för hemma-/bortalaget. Om det inte finns någon aktiv utvisning är denna en tom instans av klassen penalty.
- **Home/AwayPenalties** – Det här är en lista innehållande alla aktuella utvisningar för respektive lag, oberoende av aktiv status.

De viktigaste metoderna är:

- **AddNewPenalty** – Lägger till en ny utvisning genom att skapa en ny instans av klassen Penalty och lägger till den i listan för rätt lag.
- **RemovePenaltyWithIndex** – Tar bort en utvisning ur en lista för ett lag, med hjälp av index.
- **RemoveFinishedPenalties** – Tar bort alla slutförda utvisningar, alltså alla utvisningar där tiden löpt ut. Skapar en temporär lista med utvisningar som ska tas bort, och tar sedan bort dessa i respektive lags lista.
- **UpdateActivePenalties** – Loopar igenom respektive lags lista och sätter utvisningar med index 0 och 1 till aktiva, samt popularar

Home-/AwayPenalties med rätt utvisning, finns det ingen aktiv utvisning skapar den en tom instans av klassen Penalty i stället. Trigger sedan eventet PenaltyChanged som Game-klassen prenumerera på.

Flödet för hur en ny utvisning läggs till kan ses i figur 10 nedan och i Bilaga A.



Figur 10 - Flöde för ny utvisning (Bilaga A).

Tiden för utvisningarna använder sig av en TimeSpan, och tiden för utvisningar justeras i klassen GameTimer.

För att förstå varför den här klassen fungerar som den gör behöver man veta lite mer om reglerna för ishockey.

I Svenska Ishockeyförbundets Spelregler för ishockey 2024/2025, regel 26, står följande:

”Om en tredje spelare i något lag utvisas när två spelare i det laget redan avtjänar pågående utvisningar på utvisningsklockan, ska den tredje spelarens utvisning inte börja räknas förrän någon av de två redan pågående utvisningarna upphör.” (14)

4.2.8 Penalty

Varje instans av den här klassen representerar en utvisning. Varje utvisning har ett antal egenskaper som populeras i konstruktorn, som till exempel spelarnummer, utvisningslängd och kvarvarande tid, och kan ses i figur 11 nedan.

```
public Penalty(int playerNumber, TimeSpan penaltyLength, TimeSpan remainingTime, int team)
{
    PlayerNumber = playerNumber;
    PenaltyLength = penaltyLength;
    RemainingTime = remainingTime;
    IsActive = false;
    Team = team;
    TimeInitiated = DateTime.Now;
}
```

Figur 11 - Konstruktorn för klassen Penalty

4.3 Utveckling av kontroll och display för matchklocka

För att kunna kontrollera matchklockan skapades en kontroll-klass (GameContrller). Även en display-klass (ConsoleDisplay) skapades för att kunna

visa klockans information i konsolen, dessa två klasser är egentligen fristående och hör inte till matchklockans logik, utan dessa två kan med fördel i framtiden bytas ut mot ett grafiskt gränssnitt.

4.3.1 ConsoleDisplay – Visar den aktuella informationen

Den här klassen skapas och får med sig instansen av Game-klassen. Den prenumererar på eventet UpdateGame, och uppdaterar informationen i konsolen varje gång eventet triggas. Den visar, aktuell tid och period, samtliga utvisningar och poängställning, samt kontroller för att styra matchklockan. Den formaterar även tiden vid output för att visa minuter och sekunder, eller sekunder och tiondelar. I figur 12 nedan finns en del av en skärmdump från applikationen, hela skärmdumpen kan ses i Bilaga B.



Figur 12 - Aktuell matchinformation i konsolen (Bilaga B)

4.3.2 GameController – Kontrollera matchklockan

Den här klassen ansvarar för att kontrollera/styra matchklockan, och anropar metoder som start, stop, lägg till/ta bort mål, lägg till/ta bort/ändra utvisningar, med mera.

En instans av klassen skapas upp i Program.cs där den också får med sig en instans av klassen Game. Klassen innehåller endast en metod som är en while-loop som körs som en asynkron "Task". Den lyssnar på tangenttryckningar från användaren med hjälp av "Console.ReadKey", och i en "switch-sats" så anropas olika metoder beroende på vilken tangent som trycktes in. Metoden heter "ListenToKeyPress" och anropas med await från Program.cs. (16) (17)

```
case ConsoleKey.U:

    if (!game.GameClock.ActiveTimer.IsRunning)
    {
        Console.WriteLine("Lägg till ny utvisning\n");

        (team, cancel) = ConsoleDialogs.ChooseTeam();
        if (cancel)
            break;

        int playerNumber = ConsoleDialogs.SetPlayerNumber();

        (newPenaltyTime, cancel) = ConsoleDialogs.ChoosePenaltyTime();
        if (cancel)
            break;

        game.GamePenalties.AddNewPenalty(playerNumber, newPenaltyTime, team);
    }
    break;
```

Figur 13 - I switch-satsen när tangenten som trycktes var U.

Ett exempel är att knappen "U" används för att lägga till en ny utvisning, och den kan ses i figur 13 ovan. Den använder sig i sin tur av en statisk klass som heter "ConsoleDialogs".

4.3.2.1 ConsoleDialogs

Den här klassen innehåller flera olika dialoger som återanvänds på flera ställen i applikationen. Ett exempel är "ConsoleDialogs.Confirm", den här metoden frågar om Ja/Nej, och returnerar en tupplet med två booleska värden, ett för Ja/Nej och ett om användaren valde avbryt.

Ett annat exempel är "ConsoleDialogs.ChoosePenaltyTime" den här listar alla vanliga alternativ på utvisningslängder och ger även användaren en möjlighet att fylla i en egen tid. Om användare väljer att fylla i sin egen tid anropas en annan metod i samma klass – som heter TimeSpanFromMinutesSeconds och returnerar en TimeSpan med minuter och sekunder, den tar i sin tur hjälp av ytterligare en annan statisk klass som heter "ConvertInput".

4.3.2.2 ConvertInput

Den här klassen har några statiska metoder som tar in input från en användare och konverterar detta till exempelvis en double, heltal eller TimeSpan.

Metoden "ConvertToInt" tar input från en användare och försöker konvertera den till ett heltal, metoden returnerar en tupplet med heltal och ett booleskt värde beroende på om konverteringen lyckades eller inte.

Det finns även metoder för "ConvertToDouble" som returnera en double och "TimeSpanFromSeconds", "TimeSpanFromMinutesSeconds" som båda returnerar en TimeSpan. "TimeSpanFromSeconds" används oftast när tiden på matchklockan ska justeras. "TimeSpanFromMinutesSeconds" används

för att ange en exakt tid, antingen om man vill sätta matchtiden på klockan till en specifik till eller lägga till en utvisning med en exakt tid.

Den här klassen ha skapats för att man inte ska behöva upprepa felhantering av inmatning på flera platser i applikationen, utan kan hålla den logiken till en plats.

4.4 Övrig funktionalitet

4.4.1 GameJsonSerialzier

Den här klassen lyssnar på eventet GameUpdate i Game-klassen. Varje gång eventet triggas serialiseras data från eventet till en JSON-fil med hjälp av "JsonSerializer". Varje gång data serialiseras triggas ett event som skickar med den serialiserade datan. (18)

Detta möjliggör att andra klasser kan lyssna på det här eventet och ta emot aktuella matchdata i JSON-format.

4.4.2 FileManager

Den här klassen har i uppgift att skriva till en .JSON-fil som används för att kunna återställa matchen. Den lyssnar på eventet från GameJsonSerializer och detta triggas då en metod som skriver JSON-data till en fil med hjälp av den inbyggda File-klassen. (19)

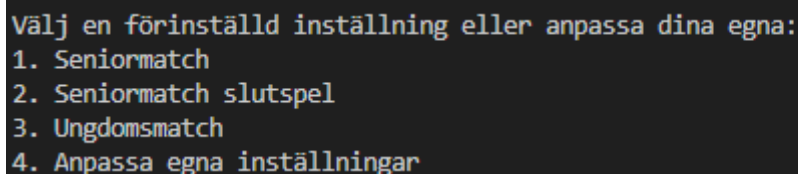
För att undvika att filen blir korrupt skrivs data först till en temporär fil, om den lyckas så skrivs den riktiga backup-filen över med den temporära filen.

4.4.3 GameStateLoader

Den här klassen har i uppgift att kolla om det finns en backup-fil för att kunna återställa matchen. Om filen finns så returneras data för att möjliggöra återställning av matchen.

4.4.4 StartUpDialog

Den här klassen har i uppgift att visa en dialog för en användare vid start av applikationen. Den kollar om det "GameStateLoader" har en fil att återställa matchen ifrån, annars ber den en användare att välja inställningar för matchen och returneras sedan dessa så att en ny match kan startas med dessa inställningar. Den anropar en annan klass som heter "GameSettingsManager" för att hämta en instans av "GameSettings" innehållande inställningar. Menyval för att välja inställningar kan ses i figur 14 nedan.



```
Välj en förinställd inställning eller anpassa dina egna:  
1. Seniormatch  
2. Seniormatch slutspel  
3. Ungdomsmatch  
4. Anpassa egna inställningar
```

Figur 14 - Menyval för att välja inställningar för match

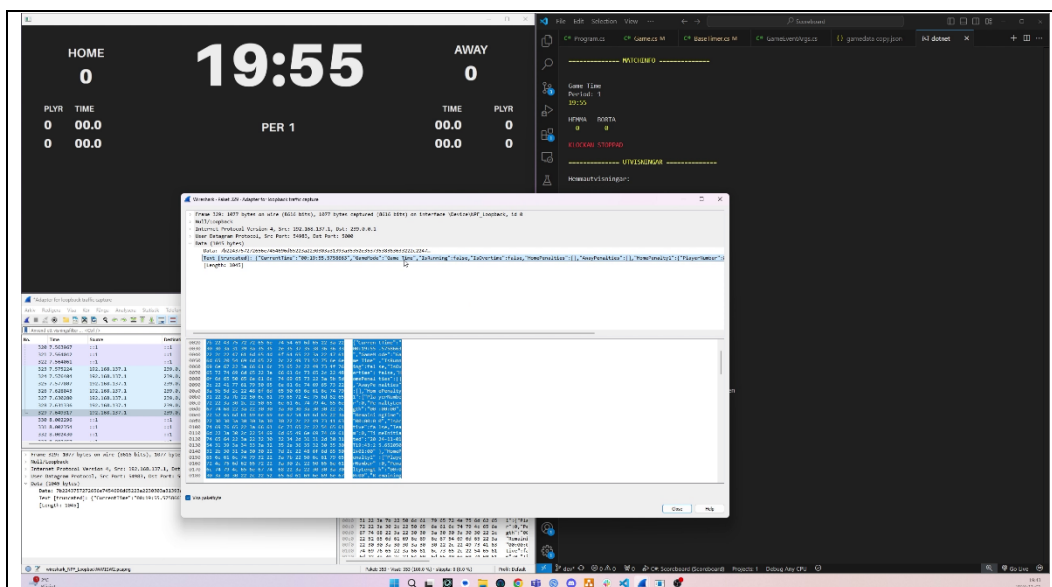
4.4.5 GameSettingsManager

Den här klassen ansvarar för att skapa en instans av "GameSettings" som används för att skapa en ny match. Den innehåller några förprogrammerade alternativ vilket täcker de flesta vanliga matchtyper i Svensk Ishockey, men har också en metod som ger användaren en möjlighet att ange sina egna inställningar.

4.4.6 UdpTransmitter

Den här klassen ansvarar för att skicka data över nätverket med hjälp av UDP Multicast. Den lyssnar på eventet i "GameJsonSerializer" och skickar en signal med informationen över nätverket. Detta gör att en resultattavla, display eller till exempel TV-produktionen kan ta emot den här datan i realtid.

När eventet är triggas så startas en asynkron "Task" som körs i bakgrunden och skickar då JSON-data över nätverket. Den använder sig av den inbyggda klassen UdpClient och dess metoder för att kunna skicka informationen över nätverket. I figur 15 nedan ses exempel på när matchklockan körs och skickar data i realtid över nätverket som sedan tas emot av en, "UDPreciever" som inte är en del av det här projektet, men visar på hur en resultattavla skulle kunna fungera, i Bilaga C finns även en länk till en video som demonstrarer detta.



Figur 15 - Skärmlapp från video, visar applikationen som skickar data över nätverket med en UDPmottagare.

4.5 Test av applikation

Applikationen testkördes på två olika enheter. En dator med Windows 11 som operativsystem och en dator med Linux (Pop!_os 24.04). Detta har fungerat utan problem, och samtliga funktioner inklusive återställning av tidigare match, samt UdpTtransmitter har fungerat.

UDPtransmittern testades med hjälp av applikationen WireShark, som är en applikation för att analysera trafik på nätverket. Även detta kan ses i videon i Bilaga C.

Funktionaliteten i applikationen har under testerna fungerat bra, jag har inte kunnat se någon märkbar fördröjning på paketen som skickas över nätverket, utan detta sker i realtid.

5 Resultat

För att ta reda på om resultatet har nåtts summeras detta nedan.

5.1 Kriterier för slutfört projekt:

- **Hantera matchtid räknad uppåt alternativt neråt. Starta, stoppa och justera tid, samt stöd för att hantera övertidsperioder med annan periodlängd. Om tiden är mindre än 1 minut ska tiden visas i sekunder och tiondelar, annars i minuter och sekunder.**

Matchklockan innehåller funktioner för att hantera matchtid, för både vanliga perioder samt övertidsperioder. Den kan även formatera tiden för att visa sekunder och tiondelar, eller minuter och sekunder. Funktionaliteten för att formatera tiden är i och för sig inte lika väsentlig i konsolen där matchklockan styrs som om den skulle visas på en resultattavla.

- **Hantera utvisningar. Funktionalitet för att lägga till, ta bort, ändra och visa utvisningar.**

Matchklockan har funktionalitet för att hantera och visa utvisningar enligt SIF:s regelverk.

- **Hantera mål. Funktionalitet för att lägga till, ta bort och visa.**

Matchklockan kan hantera och visa mål.

- **Funktionalitet för Time-out och powerbreaks, räknad nedåt till 0 sekunder.**

Matchklockan har funktionalitet för Time-out och powerbreaks, men tid som räknar nedåt till 0.

- **Funktionalitet för att hantera paustid, räknad nedåt från det totala antalet minuter till 0.**

Det finns funktionalitet som hanterar paustid, som räknar nedåt.

- **Funktionalitet för att kunna återställa tidigare match efter strömavbrott eller vid händelse att applikationen kraschar.**

Det finns funktionalitet för att återställa en match efter en oförutsedd händelse.

- **Möjlighet att skicka klockans status/signal över nätverket till displayer/skärmar, klockor i omklädningsrum och eventuellt tv-produktion via UDP Multicast.**

Det finns möjlighet att skicka klockans status/signal till andra enheter över nätverket. Om ett annat protokoll behövs finns det möjlighet att lägga till detta utan att behöva ändra något i matchklockans logik.

6 Slutsatser

Det här var ett projekt som i mitt huvud kändes som ett ganska enkelt och litet projekt, men jag märkte ganska snabbt så fort jag börjat med planeringen att det här nog var betydligt mer komplext än vad jag trott.

Efter att jag fått ner en första planering av applikationen så började jag att testa mig fram med lite olika sätt att mäta tid, vilket också gjorde att jag insåg att tid inte är helt lätt att hantera, även om C# har en del funktionalitet inbyggd för att hantera tid. TimeSpan blev en ovärderlig tillgång för att kunna hålla koll på, och justera tid.

Direkt efter att jag fått till ett helt okej sätt att mäta tiden på, gick jag sedan vidare med att lägga till funktionalitet för att starta, pausa och återuppta tiden, och det var här jag stötte på mitt första stora problem, jag kunde inte pausa tiden. Efter en del felsökande kom jag fram till att det berodde på att tidtagningen kördes på huvud tråden, samma tråd som jag försökte använda för att lyssna på mina tangenttryck. Det här var något helt nytt för mig då jag inte riktigt stött på detta problem tidigare, jag inte alls har behövt eller för den delen ens kunna hantera flera trådar när jag programmerat i JavaScript som körs i en tråd.

Jag ska inte påstå att jag greppar allt, och förstår asynkron programmering, och användandet av flera trådar fungerar i C#, utan jag lärde mig nog mycket för att få det att fungera okej i det här fallet. Jag har säkerligen gjort en hel del fel i hur man på bästa/ett bättre sätt hanterar detta, men jag har i alla fall fått doppa tån och testa på detta, vilket jag förstår att jag kommer ha fördel av när jag stöter på det här problemet nästa gång. Jag behöver definitivt fördjupa mig mer i det här.

En annan stor utmaning har varit att använda mig av objektorienterad programmering, vilket är något jag inte är vad med. Jag har mer kunskap efter projektet än vad jag hade innan, men känner att jag inte har mer än skrapat på ytan av vad det innebär. Grundtanken till varför det är användbart har jag börjat att förstå något, och en av de största fördelarna jag känner är att det blir lättare att återanvända kod, samt att den blir något mer strukturerad, och att ha möjligheten att ärva egenskaper och metoder är otroligt smidigt. Inom ramen för det här projektet har det varit väldigt användbart, då jag har tänkt att varje klass ska fungera som en liten modul, och jag kan relativt enkelt lägga till och ta bort funktionalitet i applikationen. En funktion jag skulle vilja lägga till är en klass som räknar skott på mål, och detta kan jag enkelt fixa utan att behöva ändra något i matchklockans logik, utan det enda som behövs är att en instans skapas upp i Game-klassen.

För att förbättra applikationen vidare skulle jag vilja läsa mer om hur jag kan göra olika funktioner effektivare. Jag förstår att hur jag använder mitt event, och den data som skickas med skulle kunna effektiviseras och förbättras betydligt mer, och det bör även finnas en hel del andra funktioner man kan nyttja för mer säkerhet och driftsäkerhet. Jag misstänker att det

även finns en hel del som kan optimeras kring min serialisering av JSON-data, då det görs väldigt många gånger per sekund. En sak att kolla på hade kunnat vara hur man kan serialisera endast den data som har förändrats eller liknande.

Det finns ju såklart även en hel del att tänka på kring hur jag skickar information över nätverket med UDP. Dels så är det i de allra flesta fall helt onödigt att skicka hela JSON-objektet, utan man skulle definitivt kunna banta ner information till det som endast är väsentligt för att inte skicka lika stor mängd data. I övrigt skulle det kanske vara vettigt att begränsa antalet gånger man skickar data varje sekund, till kanske max 10 gånger i sekunden för att inte bombadera nätverket med onödigt många anrop. Att lägga till fler protokoll och ge möjligheten att konfigurera IP-adresser samt övriga inställningar hade såklart också varit ett lyft.

Förutom att förbättra olika aspekter av applikationen, skulle nästa steg definitivt vara att skapa ett grafiskt gränssnitt för att kunna styra matchklockan. Att styra den med massa olika tangenter i konsolen fungerar inom ramen för det här projektet, men om den någon gång skulle användas skarpt behövs definitivt ett grafiskt gränssnitt.

En annan del att för att utveckla applikation vidare hade varit att skapa ett API, samt implementera WebSockets eller någon annan realtidslösning. Detta hade gjort att man kan styra och läsa information från väldigt många olika typer av enheter, framför allt via webben. Att bygga ett enklare gränssnitt för att visa aktuell tid, eller styra klockan via webben hade varit väldigt användbart och ett kostnadseffektivt sätt.

En sista väldigt viktig del hade såklart varit att bygga en applikation som är själva resultattavlan som ska visa informationen för publiken. Detta hade medfört att det blev fullständig lösning för en matchklocka. Men fördelen med att den skickar information över nätverket är att den kan användas med vissa befintliga resultattavlor.

För att summera så har det varit ett väldigt lärorikt projekt som har tvingat mig att utforska språket lite mer. Att få stöta på asynkron programmering och tasks har givit en ny synvinkel på hur man kan utveckla applikationer, och jag känner att jag kommer ha nytta av de i framtiden. Samtidigt så har jag bara skrapat på ytan av vad som är möjligt med C# och .NET, som är ett otroligt roligt språk och plattform att utveckla med.

7 Referenser

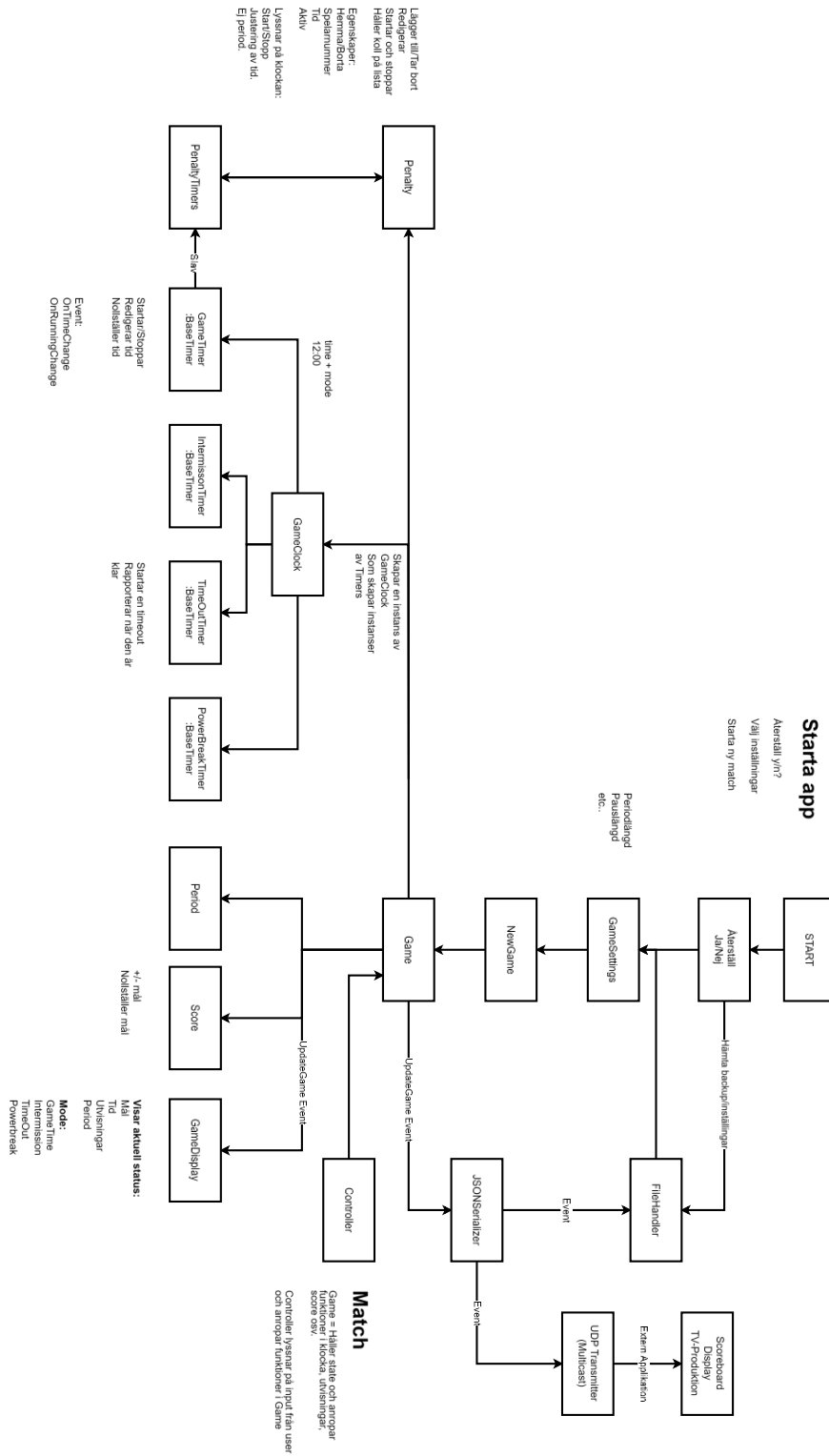
1. Svenska Ishockeyförbundet. Svenska Ishockeyförbundet | Regelbok Anläggningar. [Online].; 2024 [cited 2024 11 02. Available from: <https://www.swehockey.se/media/gpkjxlod/regelbok-anla-ggningar-24-25.pdf>.
2. Microsoft. C#. [Online].; 2024 [cited 2024 11 03. Available from: <https://dotnet.microsoft.com/en-us/languages/csharp>.
3. Microsoft. What is.NET? [Online].; 2024 [cited 2024 11 03. Available from: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>.
4. Fixa Nätet, Petter Östlund. UDP och TCP – två sätt att skicka trafik. [Online].; 2024 [cited 2024 11 03. Available from: <https://www.fixanatet.se/portar-och-nat/udp-och-tcp-tva-satt-att-skicka-trafik/>.
5. ByteHide. Async and Await in C#: Full Guide. [Online].; 2023 [cited 2024 11 03. Available from: <https://www.bytehide.com/blog/async-await-csharp>.
6. Microsoft. Task-based asynchronous programming. [Online].; 2024 [cited 2024 11 03. Available from: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming>.
7. Microsoft. Events (C# Programming Guide). [Online].; 2024 [cited 2024 11 03. Available from: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/>.
8. Microsoft. EventHandler Delegate. [Online].; 2024 [cited 2024 11 03. Available from: <https://learn.microsoft.com/en-us/dotnet/api/system.eventhandler?view=net-8.0>.
9. Wireshark. Wireshark. [Online].; 2024 [cited 2024 11 04. Available from: <https://www.wireshark.org/>.
10. Nagle D. Difference between Unicast vs Multicast. [Online].; 2024 [cited 2024 03 11. Available from: <https://netinsight.net/difference-between-unicast-vs-multicast/>.

11. sportssystem W. Sportmanualer För BASIC LED190/250/300. [Online].; 2019 [cited 2024 03 11. Available from: <https://westerstrand.se/uploads/2019/09/4303sv03.pdf>.
12. Microsoft. Stopwatch Class. [Online]. [cited 2024 11 03. Available from: <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-8.0>.
13. Microsoft. TimeSpan Struct. [Online]. [cited 2024 11 03. Available from: <https://learn.microsoft.com/en-us/dotnet/api/system.timespan?view=net-8.0>.
14. Svenska Ishockeyförbundet. Regelbok Ishockey 24/25. [Online].; 2024 [cited 2024 11 03. Available from: <https://www.swehockey.se/media/obaa1m1z/spelregler-foer-ishockey-20242025.pdf>.
15. Microsoft. Math.Max Method. [Online].; 2024 [cited 2024 11 04. Available from: <https://learn.microsoft.com/en-us/dotnet/api/system.math.max?view=net-8.0>.
16. Microsoft. Console.ReadKey Method. [Online].; 2024 [cited 2024 11 04. Available from: <https://learn.microsoft.com/en-us/dotnet/api/system.console.readkey?view=net-8.0>.
17. Microsoft. Selection statements - if, if-else, and switch. [Online].; 2023 [cited 2024 11 04. Available from: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements#the-switch-statement>.
18. Microsoft. How to write.NET objects as JSON (serialize). [Online].; 2024 [cited 2024 11 04. Available from: <https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to>.
19. Microsoft. File Class. [Online].; 2024 [cited 2024 11 04. Available from: <https://learn.microsoft.com/en-us/dotnet/api/system.io.file?view=net-8.0>.

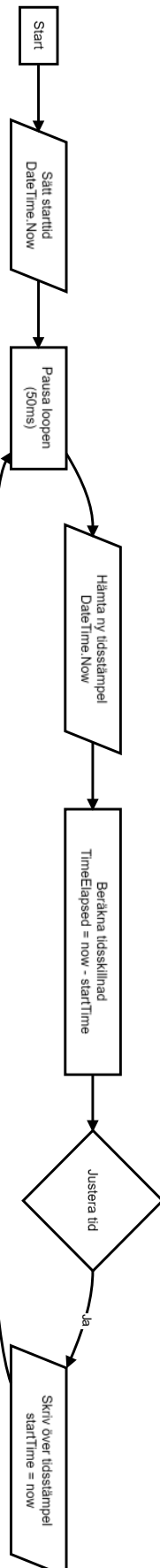
8 Bilagor

8.1 Bilaga A – Diagram och flödesscheman

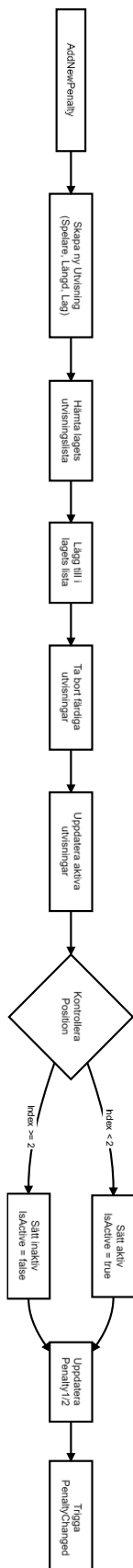
8.1.1 Diagram över applikation



8.1.2 Flödesschema för att beräkna tid



8.1.3 Flödesschema för att lägga till en ny utvisning



8.2 Bilaga B – Skärmdump från konsolen

```
===== MATCHINFO =====

Game Time
Period: 1
19:52

HEMMA    BORTA
  0       0

KLOCKAN STOPPAD

===== UTVISNINGAR =====

Hemmutvisningar:

Index  Nr    Tid    Aktiv
---
Bortautvisningar

Index  Nr    Tid    Aktiv

===== KONTROLLER =====

SPACEBAR    - Starta/Stoppa tid
H / H+SHIFT - Öka/Minska hemmamål
G / G+SHIFT - Öka/Minska bortamål

TID:
A          - Justera tid i sekunder
S          - Ändra matchtid

UTVISNINGAR:
U          - Lägg till utvisning
R          - Ta bort utvisning
E          - Ändra utvisningstid
M          - Flytta utvisning till toppen

PERIOD / TIMERS:
N / N+SHIFT - Nästa/Föregående period
I          - Paus-läge
T          - Timeout-läge
P          - Pausläge

Q          - Avsluta programmet
```

8.3 Bilaga C

Länk till video:

<https://youtu.be/EoJSBM--oSA>