Projektuppgift

DT071G – Programmering i C#.NET

Moment 5
Konsolspel: Kapitalträsk

Av: Emma Lorensson

MITTUNIVERSITETET
Avdelningen för informationssystem och -teknologi

Författare: Emma Lorensson, emlo2302@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin, år: HT, 2024

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.

Sammanfattning

Projektet gick ut på att skapa ett textbaserat, satiriskt spel i form av en konsolapplikation i C#.NET. Spelet fick namnet *Kapitalträsk* och utmanar spelaren att ta ställning till etiskt utmanande aktiviteter med målet att bli miljonär. När spelaren har uppnått en miljon i kapital vinner denne, förlorar gör man om spelarens kapital eller karma sjunker till 0.

Vid planeringen av applikationen togs ett flödesschema fram som visar spelets flöde. Därefter startades ett nytt konsolprojekt upp i C#.NET och kodens olika klasserna *Menu, Player, Game, Utilities, Scenario* och *Program* skapades. Genom hela utvecklingsarbetet har applikationen testats och felsökts efter eventuella problem som behöver lösas.

I slutändan konstaterades att en rolig spelidé tagits fram, men inte utan svårigheter längs vägen. Dels upplevdes det svårt att konstruera en bra spellogik när jag aldrig gjort det förut och dels hade jag vid en vidareutveckling av applikationen försökt strukturera kodens klasser bättre.

Överlag bedöms projektet ändå lyckat.

Innehållsförteckning

Terminologi

Nedan presenteras de akronymer som används i rapporten.

Definition av termer och förkortningar

OOP	Object Oriented Programming
.NET	Ramverk för mjukvaruutveckling
C#	Programmeringsspråk
CLR	Common Language Runtime

1 Introduktion

Denna rapport syftar till att beskriva bland annat den metodik och de tillvägagångssätt som använts för att skapa konsolspelet *'Kapitalträsk'* i C#.NET. I detta första kapitel behandlas *bakgrund och problemmotivering, syfte, avgränsningar* samt *konkreta och verifierbara mål.*

1.1 Bakgrund och problemmotivering

Kursen *'Programmering i C#.NET'* avslutas med genomförandet av ett projektarbete. Jag har beslutat mig för att skapa ett textbaserat satiriskt konsolspel med namnet *'Kapitalträsk'* som ska gå ut på att spelaren, genom att genomföra olika aktiviteter, ska bli miljonär på så kort tid som möjligt. Samtidigt måste spelaren hålla koll på sin karma och säkerställa att den ligger på plus.

Spelet ska ha tre värdeparametrar som avgör hur det går för spelaren – kapital, karma och tid spelad. Om spelarens kapital eller karma går ner till 0 är spelet över. Om spelarens kapital uppgår till 1 000 000 SEK är spelet vunnet. Tid spelad presenteras i år och månader, där en månad motsvarar tre spelade minuter.

Spelet ska inledas med att spelaren är nyinflyttad i Kapitalträsk och därför behöver någonstans att bo, tre val med olika hyressummor presenteras för spelaren, varav denne måste välja en. Detta val motsvarar spelets svårighetsgrad, då hyran kommer att dras varje spelmånad (var tredje minut) och det blir avsevärt mycket svårare att vinna spelet med en hög hyra.

När bostadsalternativ har valts presenteras spelaren för spelmenyn – där nio ”rum” finns att välja mellan. Spelaren kan välja att gå till banken, börsen, monument Profitsson, Arbetsförmedlingen, vallokalen, caféet, golfklubben och välgörenhetsgalan.

Respektive rum har sedan mellan en till tre aktiviteter som spelaren kan utföra och som på något vis påverkar dennes kapital och karma. När aktiviteten är genomförd slussas spelaren tillbaka till spelmenyn där denne väljer en ny aktivitet och så fortsätter det tills spelet är vunnet eller förlorat.

1.2 Övergripande syfte

Projektets övergripande syfte är att med objektorienterad programmering i C#.NET skapa ett textbaserat, satiriskt konsolspel vid namn ”Kapitalträsk” som utmanar spelaren att navigera genom olika ekonomiska och sociala aktiviteter med målet att bli miljonär.

1.3 Konkreta och verifierbara mål

Projektets mål är att nedan punkter ska vara uppfyllda vid färdigställande:

1. Ett tydligt flödesschema ska tas fram för spelet.
2. Objektorienterad programmering ska användas.
3. En spelmotor som möjliggör interaktiva spelmoment och som inkluderar hantering av spelarens kapital, karma och tid ska skapas.
4. Spelet ska ha ett engagerande och tydligt poäng- och belöningssystem.
5. Spelet ska bestå av minst åtta spelaktiviteter.
6. Spelet ska vara moraliskt utmanande och på ett satiriskt sätt spegla rådande samhällssystem och strukturer.

2 Teori

2.1 Objektorienterad programmering

Objektorienterad programmering är en metod för att strukturera program i klasser och samverkande objekt, snarare än funktioner och logik. Denna metod gör det enklare att hålla programkod organiserad och minskar risken för att olika delar av koden påverkar varandra oavsiktligt. Objektorienterad programmering används ofta i större projekt då metoden gör koden lättare att underhålla, återanvända och skala [1, 2, 7].

För en närmare förklaring av objektorienterad programmering kommer jag nedan att gå igenom de olika ’byggstenar’ som används för olika ändamål [1].

2.1.1 Klasser

I objektorienterad programmering fungerar en klass som en ritning för att skapa ett eller flera objekt. Klassen innehåller grundläggande värden som bestämmer vilken struktur ett objekt som skapats från klassen ska ha och specificerar dess metoder [3, 7].

En klass skapas med nyckelordet *’class’* och kan beskriva i stort sett vad som helst, men tänk dig ett husdjur. Ett husdjur kan till exempel bestå av en uppsättning attribut art, en ras, ett namn och ålder. Klassen husdjur kan då användas för att skapa flera husdjursobjekt, till exempel *’katt, perser, Kattis, 2’* eller *’hund, tax, Hundis, 5’.* Klassen skulle också kunna innehålla metoder, som är funktioner som beskriver vad objekten kan göra. Exempelvis skulle en metod för en husdjursklass kunna vara *’Leka()’,* som när den anropas påverkar energinivå eller humör hos objektet som motsvarar husdjuret [3, 7].

2.1.2 Objekt

Ett objekt är en instans skapad av en klass, som beskrivs i Kap. 2.1.1 om klasser. Objektet följer den *’ritning’* som klassen motsvarar, men definierar de attribut som särskiljer ett objekt från ett annat objekt. Se jämförelsen som görs mellan katt- och hundinstansen i Kap. 2.1.1 Klasser [2, 7].

2.1.3 Attribut

Attribut är variabler som definieras inom en klass och används för att definiera egenskaperna hos ett objekt. Attributen lagrar data som är specifika för instanser av klassen, som exemplet med husdjursklassen – där *’art’, ’ras’, ’namn’* och *’ålder’* är attribut som definierar olika egenskaper hos husdjuret [7].

2.1.4 Metoder

I objektorienterad programmering är en metod motsvarigheten till en funktion och styr därmed beteenden. Metoderna utför operationer på variabler och beskriver handlingar som objekt kan utföra. En metod kan acceptera argument som parametrar, arbeta med dem och sedan returnera ett resultat [5, 7].

Det finns ett antal olika typer av metoder som är vanliga att använda inom objektorienterad programmering.

2.1.4.1 Egenskaper

Egenskaper används för att få tillgång till och manipulera attribut på ett kontrollerat sätt, ofta genom get- och set-metoder. Genom att definiera en egenskap går det att skapa ett publikt tillgängligt gränssnitt för ett privat attribut, vilket möjliggör inkorporering av validering och andra logikregler som kontrolleras innan ett attributvärde ändras. Detta säkerställer att korrekta värden tilldelas attributen [7].

2.1.4.2 Konstruktormetoder

Konstruktormetoder används för att skapa instanser av en klass. En konstruktor kan hålla parametrar som anger de startvärden som ska tilldelas objektets attribut när de skapas. På så sätt möjliggörs en smidig initialisering av objekt som skapas utifrån klassen [5].

2.1.4.3 Instansmetoder

Instansmetoder är metoder som definieras i en klass och som endast kan anropas efter att ett objekt skapats av den specifika klassen. Dessa metoder har tillgång till objektets attribut och kan manipulera dess värden. Exemplet med *'Leka()-metoden'* i *kap 2.1.1 Klasser* är en instansmetod. En annan instansmetod i husdjursklassen skulle kunna vara *'Läte()'* som får djuret att låta på ett visst sätt beroende på vilken art det är [6].

2.1.4.4 Statiska metoder

Statiska metoder är metoder som definierar funktionalitet som inte är kopplad till en specifik instans av en klass (ett objekt). Dessa metoder tillhör snarare klassen i sig än en instans av klassen och används för att hantera funktionalitet som är gemensam för alla instanser av en klass. [6].

2.2 .NET

.NET är ett omfattande och kraftfullt ramverk som är framtaget för mjukvaruutveckling. Det är utvecklat av Microsoft för att köra applikationer på Windows operativsystem och har stöd för en mängd programmeringsspråk samt olika typer av applikationer.

Ramverket består av två huvudsakliga komponenter – Common Language Runtime (CLR) och .NET Framework Class Library. CLR ansvarar för exe-

kvering av kod skriven i något av de programmeringsspråk som stöds, medan klassbiblioteken erbjuder en stor mängd förbyggda funktioner och klasser som underlättar utvecklingen [8].

2.3 C#

C# är ett objektorienterat programmeringsspråk skapat av Microsoft och som används inom .NET ramverket. Språket har sitt ursprung från programspråksfamiljen C och är nära besläktat med andra språk som C++ och Java. Med C# kan man utveckla en mängd olika typer av applikationer, däribland mobilappar, webbappar, webbsidor, spel och mycket mer [9].

Inom ramen för nedan rubriker ges en överblick över hur en C#-applikation fungerar och är uppbyggd.

2.3.1 Namespace

Namespaces i C# används för att organisera och strukturera kod. Det finns två huvudsakliga användningsområden:

1. **Organisera .NET-klasser:** Namespaces hjälper till att hålla ordning på klasser och typer som finns i .NET-biblioteken. Ett exempel är *'System.Console.WriteLine("Katten jamar")* där *System* motsvarar namespacet, medan *Console* är en klass inuti detta namespace. Genom att ange *using System;* överst i filen behöver namspacet inte skrivas varje gång en klass inuti det används [12].

2. **Skapa egna namespaces:** Det går att deklarera egna namespaces för att undvika namnkonflikter i större projekt och hålla koden organiserad. Ett namespace deklareras med nyckelordet *namespace.* Till exempel *namespace Husdjursappen {}* [12].

2.3.2 Typsystem

C# är ett strikt typat programmeringsspråk, vilket innebär att samtliga variabler och konstanter måste ha en förbestämd datatyp. För att definiera en metod krävs dessutom att varje parameter har en specificerad typ, samt att metodens namn och returtyp är fastställda. Det innebär att typen bestäms redan när koden skrivs och inte när programmet körs [11].

En fördel med strikt typning är att det inte går att blanda olika typer hur som helst, vilket minskar risken för oväntade fel i koden [11].

2.3.3 Main()-metoden

Main()-metoden motsvarar startpunkten i en C#-applikation. Det är denna metod som körs först när programmet startas och därmed hanterar hela programmets exekvering. I *Main()* kan man anropa andra metoder, skapa objekt och ange programmets logik. När *Main()-metoden* avslutas så avslutas också programmet. Som praxis är metoden placerad i filen Program.cs. [10].

Det finns ett antal regler som definierar *Main()-metodens* utformning. Nedan listas några av dem:

- Metoden måste deklareras inuti en klass eller struktur [10].
- Metoden måste vara statisk [10].
- Metoden kan ha någon av följande returtyper: *void, int, Task* eller *Task<int>* [10].
- Metoden kan deklareras både med och utan en string[]-parameter för att ta emot kommandoradsargument [10].

2.3.4 Klasser

I C# deklareras klasser med en åtkomstmodifierare följt av nyckelordet *'class'* följt av ett unikt klassnamn, exempelvis *'public class Husdjur {}'.* Det är vanligt att strukturera enskilda klasser i egna filer för att hålla en tydlig struktur i projektet [13].

2.3.5 Objekt

För att skapa ett objekt av en klass i C# används nyckelordet *'new'* följt av klassens namn. En deklaration av ett objekt kan därmed se ut enligt följande: *'Husdjur _husdjur = new Husdjur'.* En annan metod är att skapa en objektreferens i inledningen av filen där objektet är tänkt att skapas. En objektreferens deklarerar variabeln inom vilken objektet sedan instansieras, vilket innebär att objektreferensen är *null* vid start. Ett exempel på en objektreferens kan vara *'Husdjur _husdjur;'.* Därefter instansieras objektet genom *'_husdjur = new Husdjur'* [13].

2.3.6 Egenskaper

I C# används egenskaper för att kapsla in data och skapa kontrollerad åtkomst till instansvariabler. Med get- och set-metoder kan man både hämta och sätta värden samtidigt som man möjliggör validering på dessa värden [14]. Ett exempel på hur man kan deklarera en egenskap med get-/setkod skulle kunna vara följande:

```
Private string _ras;
public String Ras
{
Get {return _ras}
Set {
If (!string.IsNullOrEmpty(value))
_name = value;
}
}
```

Men man kan också definiera egenskapen som självimplementerande vilket förenklar koden när ingen logik behövs [14].

En självimplementerad egenskap skulle kunna deklareras enligt följande:
'public string Ras {get; set;}'.

2.3.7 Konstruktorer och initialisering

I C# skrivs en konstruktor utan returtyp och med samma namn som klassen. Om vi tar husdjursexemplet så skulle en konstruktor kunna se ut enligt följande [13]:

```
public Husdjur(string art, string ras, string namn, int ålder)
{
Art = art;
Ras = ras;
Namn = namn;
Ålder = ålder;
}
```

2.3.8 Metoder

Metoder i C# kan vara antingen instanser eller statiska och de används för att definiera beteenden eller funktionalitet för objekt eller klasser. Instansmetoder kräver att ett objekt först skapas från en klass för att kunna anropas. Statiska metoder tillhör en specifik klass i sig [15].

För att anropa en instansmetod i C# används punktnotationer efter objektnamnet och för att anropa en statisk metod används klassnamnet [15].

2.3.9 Tidmätning

2.3.9.1 Stopwatch-klassen

I C# finns en fördefinierad klass vid namn Stopwatch som kan användas för att mäta förfluten tid under ett specifik tidsintervall. För att mäta ett tidsintervall används metoderna *'Start()'* och *'Stop()'*. För att hämta tiden kan man använda en egenskap vid namn *'Elapsed'* som returnerar ett *'TimeSpan'-objekt.* Det går också att återställa stopwatchen med metoden *'Reset()'*.

2.3.9.2 Timer-klassen

Timer-klassen kan användas för att skapa en timer som löper i bakgrunden av ett program och anropar en metod enligt ett förinställt tidsintervall.

3 Metod

I detta kapitel beskrivs vilka metoder som kommer att användas för att uppfylla de mål som angetts i kap. 1 Introduktion.

3.1 Flödesschema

För att planera applikationen kommer ett flödesschema skapas i DRAWIO där det tydligt framgår vilka delar av spelet som leder var och vilka beslut spelaren behöver ta ställning till. Flödesschemat fungerar som en visuell representation av spelflödet och hjälper till att definiera logiken bakom spelmekaniken och interaktionerna.

3.2 Utvecklingsprocess

När flödesschemat är färdigställt kommer ett nytt projekt att startas upp i Visual Studio 2022, och utvecklingen av programkoden kan starta. För att säkerställa en välstrukturerad filhantering och kodstruktur kommer det att genomföras research kring hur spelapplikationer vanligtvis organiseras.

3.3 Felsökning

För att hitta och lösa fel i programmet kommer Visual Studios inbyggda debugging-verktyg att användas.

3.4 Versionshantering

Projektet kommer att versionshanteras med hjälp av Git och GitHub via Visual Studios inbyggda gränssnitt för detta.

3.5 Lista över utvecklingsmiljö och verktyg

De system, program och tjänster som kommer att användas för att utveckla konsolspelet är följande:

- **Operativsystem:** Windows 11
- **Verktyg för flödesschema:** DRAWIO
- **Kodeditor:** Visual Studio 2022
- **Versionshantering:** GitHub

4 Konstruktion

4.1 Planering

Projektet inleddes med att utveckla en övergripande plan för spelet, där ton, mål och spelets logik definierades. Det beslutades att tonläget i spelet skulle vara kritiskt och satiriskt, med fokus på att belysa och ifrågasätta samhällsstrukturer och system utifrån skaparens perspektiv. Denna ton skulle genomsyra både berättelsen och de interaktiva momenten i spelet för att skapa ett tankeväckande spel.

Som en motkraft till ovan beslutades också att spelets mål skulle vara relativt simpelt. Målet fastställdes till att spelaren ska klara att utöka sitt kapital till 1 000 000 SEK utan att dennes karmapoäng hamnar på 0. Om kapital eller karmapoäng går ner till 0 betyder det att spelet är över.

För att uppnå målet beslutades att spelaren ska ställas inför ett antal moraliska dilemman som på något sätt påverkar både kapital och karma, positivt eller negativt. Samtliga dilemman presenteras inom ramen för en fiktiv plats i den fiktiva staden *'Kapitalträsk'.*

4.2 Skapande av flödesschema

När den övergripande planen var färdigställd och alla de stora besluten fattade övergick arbetet till att påbörja utvecklingen av ett flödesschema (se figur 1 för miniatyr eller bilaga 1 för större version). Flödesschemat skapades i verktyget DRAWIO i Visual Studio Code.

Flödesschemats framtagande och beslut kring speldetaljer togs fram parallellt, genom en iterativ process där aktiviteter justerades, flyttades runt eller togs bort tills helheten kändes logisk och väl avvägd.

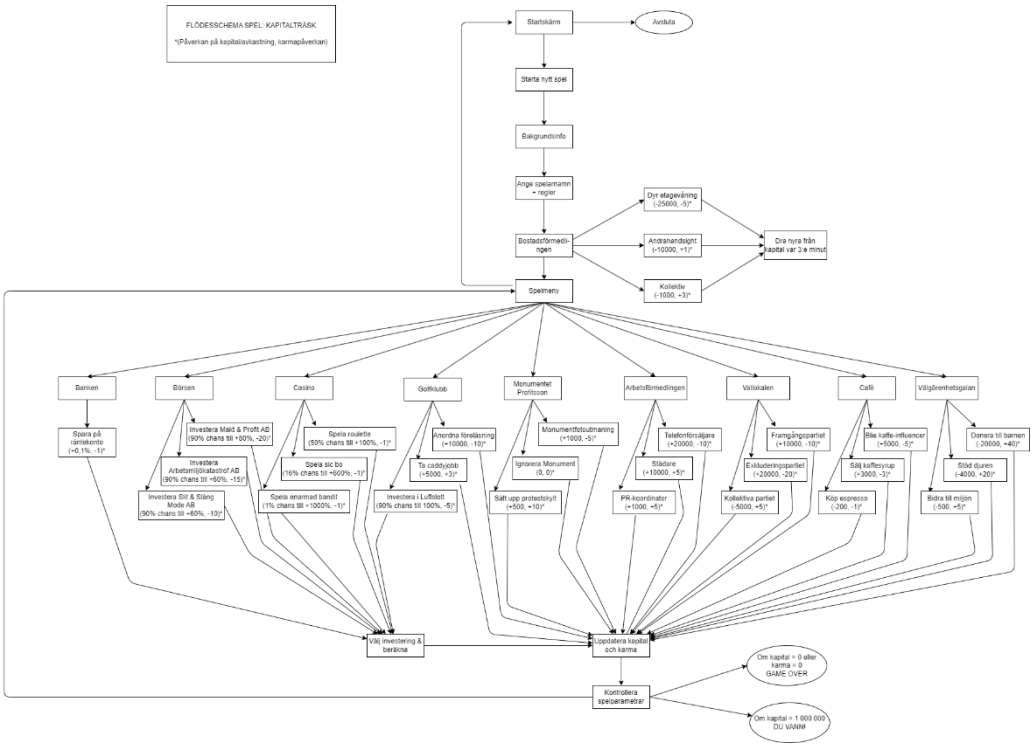
Den slutgiltiga versionen av flödesschemat landade i ett spel där spelaren först möts av en välkomstskärm där denne kan välja mellan att starta ett nytt spel eller avsluta spelet. Om *'starta nytt spel'* väljs leds spelaren till en skärm med bakgrundsinformation som krävs för att förstå spelet och därefter följer en skärm där spelaren ombeds ange sitt namn och spelets regler presenteras.

Vid nästkommande skärm startar spelet på riktigt då spelaren ombeds att välja vilket boende denne vill ha i kapitalträsk. Boendevalet representerar svårighetsgraden på spelet då hyra för valt boende dras löpande, var tredje minut, genom hela spelupplevelsen.

När spelaren har valt sitt boende, leds denne till en skärm som visar spelmenyn. Från spelmenyn kan spelaren välja att gå till nio olika platser, där varje plats erbjuder aktiviteter som påverkar både kapital och karma. När spelaren valt vilken aktivitet denne vill utföra visas antingen en skärm som

frågar spelaren hur mycket denne vill investera, eller en skärm som direkt talar om vilken påverkan valet hade på kapital och karma. I fallet där spelaren ombeds ange vad denne vill investera följer skärmen om påverkan på kapital och karma därefter. Efter varje val kontrolleras kapital och karma för att se om spelaren vunnit, förlorat eller om spelet ska fortsätta.

Spellogiken loopas därefter från spelmenyn ner till kontroll av spelparametrar tills spelet är antingen vunnet eller förlorat.



Figur 1. Miniatyr över flödesschema till spelet Kapitalträsk

4.3 Utveckling av applikation

När flödesschemat var klart inleddes arbetet med att skriva kod för spelet. Ett nytt projekt för en konsolapplikation startades upp i Visual Studio 2022 och de klassfiler som var tänkta att användas skapades. Detta resulterade i fem klassfiler utöver startklassen Program, som är placerad i filen Program.cs. Klasserna döptes till Game, Menu, Player, Scenario och Utilities.

4.3.1 Menu-klassen

Menu-klassen hanterar menysystemet i spelet och fungerar som ett gränssnitt för spelaren där denne kan interagera med olika delar av spelet och ta ställning till nya scenarier.

Klassen inleds med en privat objektreferens till spelaren som därefter instansieras i en konstruktor för Menu-klassen. Detta för att spelarinstansen ska kunna användas i Menu-klasen och alltid vara tillgänglig när menyfunktionerna behöver interagera med spelkaraktären. Därutöver har klassen två metoder *'ShowMainMenu'* och *'GameMenu'* som representerar olika delar av menysystemet.

4.3.1.1 Metoder

ShowMainMenu()
Metoden *'ShowMainMenu'* hanterar huvudmenyn i spelet och visar två val för spelaren – att starta ett nytt spel eller att avsluta programmet. Inledningsvis rensar metoden konsolfönstret genom *Console.Clear()-metoden* för att därefter skriva ut välkomsttext och menyn till spelaren. Därefter tar den in spelarens val och utför en switch-kontroll baserat på svaret. Om spelaren svarar ”1” startas ett nytt spel genom att anropa Game-klassens metod *GameIntro()* (Se Kap. 4.3.3 Game-klassen) som ger en spelintroduktion och om spelaren svarar ”2” anropas en metod för att avsluta programmet. Switch-satsen innehåller också en default som skriver ut till konsolen att valet är ogiltigt om något annat än 1 eller 2 anges av spelaren.

GameMenu()
Metoden *'GameMenu'* hanterar spelmenyn och tillhandahåller de platser som spelkaraktären kan välja att besöka i spelet. Först i metoden anropas metoden ShowPlayerInfo (se Kap. 4.3.2 Player-klassen) med ett argument som representerar spelaren. Därefter listas de nio platser som spelkaraktären kan välja att besöka, samt ett val för att gå till spelets startmeny, det vill säga anropa *'ShowMainMenu'.*

4.3.2 Player-klassen

Player-klassen representerar spelaren i spelet och innehåller attributen *'Name', 'Capital',* och *'Karma',* som hanteras med tillhörande publika getters och setters. Utöver detta innehåller klassen en privat referens till en spelinstans av typen Game, vilket möjliggör samspel mellan spelaren och spelet. Attributen och referensen till spelet initieras i en publik konstruktor för klassen, som tar spelarens namn, initiala kapital, initiala karma och en spelinstans som argument.

4.3.2.1 Metoder

Klassen innehåller tre huvudsakliga metoder: *UpdateCapital, UpdateKarma* och *ShowPlayerInfo.* Dessa metoder ansvarar för att uppdatera spelarens tillstånd och visa relevant information om spelaren.

UpdateCapital()
Den första metoden, *UpdateCapital,* används för att justera spelarens kapital efter genomförandet av en aktivitet. Metoden tar ett argument som re-

presenterar den summa som spelaren vunnit eller förlorat på aktiviteten och uppdaterar attributet *Capital* därefter. Om kapitalet skulle hamna under noll, säkerställer en if-sats att det istället sätts till noll, då spelet är förlorat.

UpdateKarma()
Den andra metoden, *UpdateKarma,* fungerar på liknande sätt som föregående beskriven metod, men den hanterar istället spelarens karma. Metoden tar ett argument som anger förändringen i karma efter en aktivitet och adderar denna till attributet karma.

ShowPlayerInfo()
Den sista metoden, *ShowPlayerInfo,* är en statisk metod som ansvarar för att skriva ut spelarens status till konsolen. Denna metod tar ett argument motsvarande en spelinstans och använder sid av *Console.Write* och *Console.WriteLine* för att visa spelarens namn, saldo och karma i grön text. Vidare anropas spelklassens metod *PrintTotalTimePlayed* för att visa den totala spelade tiden (Se Kap. 4.3.3 Game-klassen). Metoden innehåller också logik för att varna spelaren om värdena för *Capital* eller *Karma* börjar bli låga. Detta görs genom if-satser som skriver ut varningstext i rött om något av dessa attribut understiger angivna värden.

4.3.3 Game-klassen

Game-klassen är den största klassen och den ansvarar för att hantera spelprocessen, interaktionen med spelaren och koordineringen av de olika spelalternativen. Klassen inleds med privata referenser till instanser av klasserna *Player* och *Menu.* Därefter instansieras en tom lisa *(List<Scenario>),* en *Stopwatch,* en *Timer* samt två tomma variabler som är avsedda för att lagra spelarens hyra och sammanlagd speltid i minuter.

Klassen innehåller även en offentlig egenskap, *Player,* som använder en getter för att komma åt information om den aktiva spelaren. Konstruktorn för klassen ansvarar för att initialisera flera viktiga komponenter i spelet. Den anropar metoden *InitializeScenarios* (Se Kap. XXXX), skapar en ny instans av *Stopwatch* och initierar en ny menyhanterare genom att skapa en instans av Menu-klassen.

4.3.3.1 Metoder

Start()
Eftersom Game-klassen sköter flera olika delar av spelet innehåller den också ett relativt stort antal metoder. Den första metoden som skapades är *Start(),* som anropar Menu-klassens metod *ShowMainMenu()* (Se Kap. 4.3.1 Player-klassen) och skickar med den aktuella instansen av Game-klassen.

GameIntro()
Därefter skapades metoden *GameIntro()* som anropas från Meny-klassens *ShowMainMenu()* (Se Kap. 4.3.1 Menu-klassen). *GameIntro()* rensar konsolen och skriver ut spelets bakgrund till konsolen med hjälp av den inbyggda

metoden *Console.WriteLine().* Spelaren uppmanas i slutet av texten att trycka Enter för att fortsätta varav konsolen läser tangenttryckningen och anropar nästa metod – *StartNewGame().*

StartNewGame()
StartNewGame()-metoden rensar konsolen och uppmanar spelaren att ange sitt namn. Namnet tas emot med *Console.Readline()* och lagras i en variabel av typen *string*. Därefter skapas en ny instans av spelaren genom att ett nytt *Player-objekt* initialiseras via ett konstruktorsanrop och skickar med spelarens namn, ett startkapital på 70000, ett karmavärde på 70 och den aktuella instansen av *Game.* Metoden innehåller också felhantering för att kontrollera så att spelaren matar in ett giltigt värde när denne ombeds att ange sitt namn. Om spelaren inte matar in någonting och klickar på Enter ombeds denne att försöka igen. När spelaren matat in ett giltigt namn anropas metoden *WelcomePlayer().*

WelcomePlayer()
WelcomePlayer()-metoden rensar konsolen och välkomnar spelaren med dennes namn genom att hämta detta från *Player-objektet.* Därefter skrivs spelets regler ut till konsolen och spelaren ombeds att trycka Enter för att gå vidare. *Console.ReadLine()* fångar upp spelarens tangenttryck och startar den *Stopwatch* som tidigare initierats. Därefter anropas metoden *StartTimer()* följt av metoden *StartScenario(0).*

StartTimer()
Metoden *StartTimer()* initierar och startar en timer som är inställd på att ticka var tredje minut (180 000 millisekunder). Varje gång timern når tre minuter anropas metoden *DrawRent()* som ansvarar för att dra spelarens hyra. Därefter nollställs timern och startas om.

DrawRent()
Metoden *DrawRent()* ansvarar för att hantera spelarens hyresbetalningar och är utformad för att kunna kopplas till en *Elapsed-händelse* från spelets timer. Detta innebär att metoden automatiskt kan anropas vid regelbundna, bestämda tidsintervall, vilket ger en känsla av att tiden fortskrider i spelet. Timern konfigureras i metoden *StartTimer()* som triggar *DrawRent()* var tredje minut med hjälp av *ElapsedEventArgs* som parameter. Inuti metoden subtraheras spelarens befintliga kapital med den summa som lagrats i variabeln *monthlyRent.* Spelaren informeras med röd text i konsolen varje gång hyran dragits och därefter kontrolleras spelarens status med hjälp av metoden *CheckGameStatus().*

StartScenario()
Metoden *StartScenario(int scenarioIndex)* rensar inledningsvis konsolen för att därefter anropa Player-klassens metod *ShowPlayerInfo(_player)* (Se Kap. 4.3.2 Player-klassen). Efter detta anrop följer en if-sats som kontrollerar att scenariots index är giltigt, det vill säga att det är över eller lika med noll

och att det är under eller lika med antalet scenarios som finns i scenariolistan. Om scenarioindexet är ogiltigt skrivs ett felmeddelande ut i konsolen och därefter avbryts metoden. Om scenarioindexet är giltigt fortsätter metoden genom att hämta det valda scenarioobjektet via *_scenarios[scenarioIndex]* och lagra det i en variabel. Därefter ändras förgrundsfärgen i konsolen till cyan och metoden skriver ut scenarioobjektets parametrar för lokalisation och beskrivning av frågan. Direkt efter anropas scenarioklassens metod *ShowImpacts()* (Se Kap. 4.3.4 Scenario-klassen) och spelaren uppmanas att ange siffran för det scenario denne vill besöka därnäst. Istället för att direkt läsa ut spelarens inmatning ifrån denna metod anropas en annan metod *GetValidOption(Scenario)* som returnerar ett giltigt index för scenarion vilket vi lagrar i en variabeln *optionIndex* i metoden vi befinner oss. Därefter följer en if-sats som kontrollerar om scenariot har en fast procentuell avkastning genom att anropa metoden *HasReturn(scenario, optionIndex),* om detta är fallet anropas metoden *HandleInvestmentScenario(scenario, optionIndex)* annars anropas metoden *HandleNonReturnScenario(scenario, optionIndex, scenarioIndex).*

GetValidOption()
Metoden GetValidOption(Scenario) ansvarar för att kontrollera användarens inmatning vid val av scenario och returnera ett giltigt scenarioindex. Metoden inleder med att deklarera en variabel som är tänkt att lagra scenarioindexet och tilldela den ett ogiltigt värde på -1. Därefter används en while-loop som upprepas så länge spelarens inmatning inte matchar ett giltigt index inom scenariolistans alternativ.

Användarens inmatning läses in med *Console.ReadLine()* och lagras i en variabel för att därefter kontrolleras med två olika if-satser. Den första if-satsen kontrollerar om de är värdet ”0” som lagrats i variabeln och om det stämmer anropas Menu-klassens metod *GameMenu(this)* för att ta spelarens tillbaka till huvudmenyn, därefter returnerar metoden -1 för att indikera att inget giltigt val gjorts.

Den andra if-satsen använder *int.TryParse* för att kontrollera om det inmatade värdet är ett heltal inom intervallet 1 till antalet alternativ i scenariolistan. Om det stämmer så justeras indexet med *optionIndex*—då listans index är nollbaserat. Om inmatningen är ogiltig återställs *optionIndex* till -1 och spelaren uppmanas att försöka igen. Loopen fortsätter tills ett giltigt val görs och därefter returneras det giltiga indexet till den anropande metoden.

HasReturn()
HasReturn(Scenario, int)-metoden är en boolesk och används för att kontrollera om det valda scenariot har en fast procentuell avkastning kopplad till sig eller inte. Metoden tar två parametrar, ett scenario-objekt och ett index för det scenario spelaren valt. Funktionaliteten i metoden består därefter av en if-sats som kontrollerar om det finns några *null-värden* i listan *Returns* i scenariot och om så är fallet returneras *false.* Om listan innehåller

giltiga värden undersöker metoden om det valda alternativet har faktiska värden i sin lista med *.HasValue* och metoden returnerar då *true.*

HandleInvestmentScenario()
Metoden *HandleInvestmentScenario()* används för att hantera de scenarier där spelaren kan välja att investera pengar och få en procentuell avkastning tillbaka. Metoden tar två parametrar – ett scenario-objekt och ett index som motsvarar spelarens val *(optionIndex).* Inledningsvis rensas konsolen för att därefter visa spelarens aktuella spelparametrar med Player-klassens metod *ShowPlayerInfo(_player).* Därefter tillfrågas spelaren hur mycket denne vill investera genom en *Console.WriteLine()* som skriver ut frågan och spelarens val via *Scenario.OptionName[optionIndex].* En variabel initieras utan tilldelat värde för att senare lagra den summa spelaren väljer att investera.

För att kontrollera om spelaren valt ett giltigt värde att investera används en while-loop som körs tills spelaren har angett ett belopp som inte överstiger dennes befintliga kapital och som överstiger 0. Om kontrollen misslyckas får spelaren ett felmeddelande och uppmanas försöka igen. Efter lyckad kontroll skapas en ny instans av C#:s inbyggda Random-klass via kodstycket *"Random random = new Random()".* Detta åtföljs av en boolesk variabel som returnerar *true* om spelarens valda index är 1 eller om *random.NextDouble()* genererar ett flyttal som är under 0.75, vilket motsvarar 75% chans att få avkastning på den investerade summan.

Därefter kallas Player-klassens metod *ShowPlayerInfo(_player)* igen och spelarens investering skrivs ut till konsolen tillsammans med en text som antingen förklarar för spelaren hur hög avkastning denne fick på investeringen eller hur mycket denne förlorade. Logiken styrs av en if-sats som skriver ut ovannämnda alternativ beroende på om den booleska variabeln som beskrivs ovan returnerar *true* eller *false.*

Därefter lagras karmapåverkan av alternativet i en variabel och spelarens karma och kapital uppdateras och kontrolleras så att de inte är noll via metodern *UpdateCapital(), UpdateKarma() och CheckGameStatus().*

Spelaren uppmanas till sist att trycka på valfri tangent för att fortsätta och leds därefter tillbaka till spelmenyn via Menu-klassens metod *GameMenu().*

HandleNonReturnScenario()
Metoden *HandleNonReturnScenario()* används för att hantera både casinoscenarion och scenarion med fast ekonomisk påverkan. Metoden tar tre parametrar – ett scenarioobjekt, ett index som motsvarar spelarens val *(optionIndex)* och ett index som motsvarar scenariots index *(scenarioIndex).*

Likt tidigare beskriven metod inleder även denna med att kalla på playerklassens metod *ShowPlayerInfo(_player).* Därefter följer en if-sats som kontrollerar om scenariots index är lika med 8, vilket indikerar att spelaren

valt att besöka casinot. Om detta villkor uppfylls körs en switch-sats som genom att kontrollera spelarens val i *optionIndex* omdirigerar spelaren till motsvarande alternativs scenario via metoderna *PlayRoulette(), PlaySicBo()* och *PlayBandit()*.

Om scenariots index inte är lika med 8 hoppar metoden ut ur if-satsen och hanterar ett scenario med ett fast belopp som ekonomisk påverkan. Metoden kontrollerar då att listorna *FinancialImpacts* och *KarmaImpacts* innehåller giltiga värden och hämtar påverkan baserat på spelarens valda alternativ (se figur 2, figur 3).

```
double financialImpact = scenario.FinancialImpacts != null && optionIndex < scenario.FinancialImpacts.Count
    ? scenario.FinancialImpacts[optionIndex] ?? 0 : 0;
```

Figur 2. Variabel som lagrar och kontrollerar datan för fast ekonomisk påverkan för ett specifikt scenario.

Koden i figur 2 bestämmer spelarens ekonomiska påverkan baserat på dennes valda alternativ i ett scenario. Inledningsvis kontrolleras att listan *FinancialImpacts* inte är *null* och att värdet i *optionIndex* ligger inom listans gränser. Om dessa villkor uppfylls hämtas värdet från *FinancialImpacts-listan* för det indexerade val som lagras i *optionIndex.* Eftersom värdena i listan kan vara av typen *nullable* används en nullkollisionsoperator och standardvärde (”?? 0 : 0”) för att sätta värdet till 0.

```
int karmaImpact = scenario.KarmaImpacts != null && optionIndex < scenario.KarmaImpacts.Count
    ? scenario.KarmaImpacts[optionIndex] : 0;
```

Figur 3. Variabel som lagrar och kontrollerar datan för karmapåverkan för ett specifikt scenario.

Koden i figur 3 fungerar nästan på samma sätt som den i figur 2. Men här hanteras istället spelarens karmapåverkan och värden hämtas från listan *KarmaImpacts* för det alternativ spelaren valt.

Efter att detta gjorts uppdateras och kontrolleras spelarens parametrar för kapital och karma på samma sätt som beskrivits under rubriken för metoden *HandleInvestmentScenario().* Därefter skrivs spelarens val ut till skärmen, vilken karma- och ekonomisk påverkan det hade och en uppmaning att trycka på valfri tangent för att fortsätta ut på skärmen. Därefter anropas Menu-klassens metod *GameMenu()* för att slussa spelaren tillbaka till spelmenyn.

PlayRoulette()

PlayRoulette()-metoden hanterar en simulering av ett roulettespel där spelaren kan satsa pengar när denne ”besöker” casinot. Metoden inleder med att skriva ut spelarens information högst upp, därefter tillfrågas spelaren om denne vill satsa på rött ([1]) eller svart ([2]) och dennes svar lagras i en variabel genom *Console.ReadLine().* Därefter följer en if-sats som kontrollerar att spelaren angett antingen siffran ett eller två, har någon annan siffra angetts skrivs ett felmeddelande ut i konsolen.

Om ett giltigt värde angetts och tilldelats variabeln fortsätter metoden genom att skriva ut en fråga om vilken summa spelaren vill satsa och spela-

rens svar lagras i ytterligare en variabel. Spelarens inmatning kontrolleras sedan för om det är ett positivt värde och går att omvandla till ett *double-värde* via en if-sats med hjälp av *TryParse()*. Om dessa villkor uppfylls körs ytterligaren en if-sats som kontrollerar spelarens inmatning understiger eller är lika med dennes befintliga kapital. Om villkoret uppfylls skapas en ny instans av klassen *Random*. Därefter anropas metoden *Next(1, 3)* på *rand* för att generera ett heltal inom intervallet 1 till 2 och resultatet lagras i en variabel. På så sätt avgör slumpen vilken färg som blir den vinnande.

Variabeln med det slumpmässiga heltalet kontrolleras sedan mot variabeln med spelarens inmatning i ytterligare en if-sats. Om variablenas värden matchar anropas metoderna *UpdateCapital(investerad summa * 2)* och *UpdateKarma(scenario.karmaImpact)* från Player-klassen. Därefter kontrolleras spelarinstansens status med metoden *CheckGameStatus().* Slutligen kallas Player-klassens metod *ShowPlayerInfo()* tillsammans med två *Console.WriteLine()* som skriver ut att spelaren vunnit det dubbla av vad denne investerat och hur stort dennes totala kapital nu är.

Om spelarens inmatning inte överensstämmer med det slumpmässiga talet anropas metoden *UpdateCapital() istället* med *-minus investerad summa* som parameter istället. Karman uppdateras på samma sätt oavsätt vinst eller förlust. Utskrifterna i konsolen talar istället om hur mycket kapital spelaren förlorat.

Spelaren uppmanas därefter att trycka på Enter för att fortsätta.

PlaySicBo()
Metoden *PlaySicBo()* används för att simulera ett SicBo-spel där spelaren slår en tärning och vinner sex gånger sin insats om denne satsat på det nummer tärningen visar. Metoden fungerar nästan likadant som metoden *PlayRoulette()* med den skillnaden att spelaren istället får välja ett nummer mellan 1 till 6 och och det randomiserade nummer som skapas med *rand.Next()* kan också landa på siffrorna 1 till 6. Om variablerna för spelarens valda nummer och det randomiserade numret överensstämmer uppdateras spelarens kapital istället med sex gånger det investerade beloppet.

PlayBandit()
Metoden *PlayBandit()* används för att simulera en förenklad variant av en enarmad bandit och fungerar även den väldigt likt de två föregående beskrivna metoderna – *PlaySicBo()* och *PlayRoulette().* Skillnaden här ligger i att spelaren endast får frågan om hur mycket denne vill satsa, sedan är det alltid 1% chans att vinna 100 gånger den satsade summan. Istället för att kontrollera en matchning mellan spelare och randomiserat nummer, kontrolleras om det randomiserade numret understiger eller är lika med 0,1 med hjälp av metoden *NextDouble()* på *rand* och vid vinst uppdateras spelarens kapital med 100 gånger det satsade kapitalet.

CheckGameStatus()
Metoden *CheckGameStatus()* ansvarar för att kontrollera om spelet är Game

Over eller vunnet efter varje aktivitet som spelaren utför. Metoden tar en parameter i form av en spelarinstans och består därefter av en simpel if-sats som först kontrollerar om spelaren kapital uppnått 1 000 000 eller mer, då anropas metoden *PlayerWon().* Därefter kontrolleras om spelarens kapital är lika med eller understiger 0, då anropas metoden *GameOverCapital().* Slutligen kontrolleras om spelarens karma är lika med eller understiger 0, då anropas metoden *GameOverKarma().*

GameOverCapital()
GameOverCapital()-metoden anropas från metoden *CheckGameStatus()* när spelarens kapital är mindre eller lika med 0. Spelaren har i det fallet förlorar spelet och konsolen rensas för att därefter skriva ut ett Game Over-meddelande. Slutligen uppmanas spelaren att trycka på valfri tangent för att avsluta och därefter avslutas spelet med metoden *ExitGame().*

GameOverKarma()
GameOverKarma()-metoden anropas även den från metoden *CheckGameStatus(),* men när spelarens karma är mindre eller lika med 0. Metoden är identisk med metoden *GameOverCapital().*

PlayerWon()
PlayerWon()-metoden anropas också från metoden *CheckGameStatus(),* men i detta fall sker det om spelarens kapital är lika med eller överstiger 1 000 000. I detta fall har spelaren vunnit och ett meddelande om vinst skrivs ut i konsolen innan spelet avslutas med metoden *ExitGame().*

ExitGame()
Metoden *ExitGame()* stoppar stopwatchen med Stop()-metoden på stopwatch-instansen. Därefter skrivs den totala spelade tiden ut och spelaren uppmanas att trycka på valfri tangent för att stänga ner programmet.

4.3.4 Scenario-klassen

Scenario-klassen representerar en central del i spelets berättelse då den ansvarar för att hantera de olika scenarier spelaren kan interagera med och göra val i. Klassen används för att definiera situationer, där varje scenario representerar en fysisk plats i staden Kapitalträsk. Varje plats har en tillhörande beskrivning och en uppsättning alternativ som spelaren kan välja mellan.

Klassen innehåller de publika attributen *'Localisation', 'Question', 'OptionName', 'Options', 'KarmaImpacts', 'FinancialImpacts'* och *'Returns'* som hanteras med tillhörande getters och setters. Samtliga av attributen förutom de två förstnämnda är av typen *List<>* och används för att hantera listorna för de olika scenarierna.

Attributen initieras i en konstruktor för klassen som tar samtliga av attributen som argument. Utöver konstruktorn finns två tillhörande metoder i

klassen – *InitializeScenarios()* och *ShowImpacts()* som beskrivs vidare nedan.

4.3.4.1 Metoder

List<Scenario> InitializeScenarios()

Metoden *InitializeScenarios()* returnerar en lista innehållande samtliga av de scenarios som användaren kan interagera med i spelet. Varje scenario representeras som ett objekt av typen *Scenario* som innehåller en titel, en beskrivning, ett antal alternativ för användaren att välja mellan samt olika typer av påverkan på karma och kapital (Se figur 4). Listan är indexerad med 0-index, vilket justeras till 1-index i Game-klassens metod *GetValidOption()* och varje gång metoden kallas i spelet kallas den med scenariots index.

```
/*index 0*/
new Scenario(
    "VÄLKOMMEN TILL BOSTADSFÖRMEDLINGEN",
    "Som  nyinflyttad i Kapitalträsk behöver du någonstans att bo. Du har därför tagit dig ti

    new List <string>
    {
        "Takvåningen",
        "Andrahandstvåan",
        "Kollektivet"
    },

   new List<string>
    {
        "Slå till på en överprisad takvåning för 25,000 SEK i månaden - komplett med minimali
        "Flytta in i en andrahandstvåa på 50 kvm för 10,000 SEK i månaden. Helt okej komfort,
        "Bo i ett kollektiv med stadens mest omtänksamma själar. Där delas såväl ekologiska g
    },

    new List<int> { -5, +1, +3 }, // Karmapåverkan
    new List<double?> { -25000, -10000, -1000 }, //Ev. Fast ekonomisk påverkan
    new List<double?> { null, null, null } //Ev. Procentuell påverkan på investerat kapital

),
```

Figur 4. Scenario med ojusterat index 0.

ShowImpacts()

Metoden *ShowImpacts()* ansvarar för att strukturera och dynamiskt visa information om de olika scenarioalternativens karma- och kapitalpåverkan. Metoden visar konsekvenserna av varje alternativ på ett mer lättförståeligt sätt genom att skriva ut det i textformat.

Metoden inleds med en if-sats som kontrollerar om det bara finns ett alternativ i scenariot. Om det endast finns ett alternativ används *Console.Writeline()* för att skriva ut det alternativets titel, dess beskrivning och karma- och kapitalpåverkan. Om det däremot finns fler alternativ körs en for-loop så länge *i* är under *Options.Count*, det vill säga antalet alternativ som existerar till det valda scenariet. For-loopen innehåller i sin tur ytterligare en if-sats som kontrollerar att finansiell påverkan inte är satt till *null,* att indexet ligger inom listans gränser, samt att *i* faktiskt innehåller ett värde. Om dessa kontrollera uppfylls innebär det att den finansiella påver-

kan är en fast summa, t. ex 5000 SEK, och till alternativet skrivs då ut *"Fast ekonomisk påverkan: {impact}"* där *impact* är en variabel som lagrar *FinancialImpacts* hämtat från motsvarande listobjekt. Därefter följer en *else if* som kontrollera att *Returns* inte är null, att *Returns* har ett giltigt index och att antalet värden i *Returns* inte överstiger *i.* Det innebär att det är en procentuell ekonomisk påverkan på en summa som spelaren investerar. Spelaren väljer alltså själv hur mycket kapital denne ska få avkastning på. Ett exempel på hur det skrivs ut är: *"90% chans till avkastning om {impact*100}"* där *impact* i detta fall lagrar *Returns* från motsvarande listobjekt. Om både *Returns* och *FinancialImpacts* är *null* för scenariots alternativ hoppar if-satsen vidare till *else* som skriver ut att ekonomisk påverkan står i beskrivning. Denna gäller endast för Roulettespelen som beskrivs i Kap. *4.3.3 Game-klassen.*

4.3.5 Program-klassen

Program-klassen innehåller endast ett kort kodstycke som fungerar som startpunkten för applikationen. Den kod som är skriven inom klassen är skriven inom en *Main()-metod* vilket är den metod som alltid exekveras när ett program skrivet i C# startar (Se figur 5).

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {

        Game game = new Game();
        game.Start();
    }
```

Figur 5. Illustration av Main-metoden.

5 Resultat

Projektets övergripande mål var att skapa ett textbaserat, satiriskt konsolspel i programspråket C#.NET. Koden skulle vara väl planerad, objektorienterad och väl strukturerad.

Detta mål bröts sedan ner i mer konkreta delmål som skulle vara uppfyllda vid projektslut. Det första målet var att ett tydligt flödesschema skulle tas fram för spelet för att förenkla utvecklingen av detsamma. Ett flödesschema togs fram vid planeringsstadiet av applikationen som jag, som utvecklare, ansåg var tydligt. Däremot är det första gången ett flödesschema tas fram under utvecklarens utbildning och därför kan ingen objektiv bedömning gällande om flödesschemat är tydligt eller ej göras.

Det andra målet var att objektorienterad programmeringsstruktur skulle användas vid utvecklingen av projektet. Objektorienterad programmeringsstruktur har använts genom utvecklingen av hela projektet, men strukturen hade kunnat förbättrats ytterligare. Spelets kod är strukturerat med flera mindre klasser och en klass innehållandes en stor mängd kod. Den stora klassen, *Game()-klassen,* innehåller kod för både menyhanterings, spelflöde, tidshantering, kontroller av avkastning etcetera. Objektorienteringen är skulle därmed kunna bedömas inte vara helt ändamålsenlig. I slutändan skulle jag vilja hävda att målet är uppfyllt, men att förbättringar vore önskvärda.

Det tredje målet var att en spelmotor som möjliggör interaktiva spelmoment skulle skapas och som inkluderar hantering av spelarens kapital, karma och tid. Detta mål anses vara uppfyllt då spelet blir interaktivt med hjälp av dessa spelparamterar. En effektiv tidshantering har skapats där spelarens hyra dras varje fiktiv månad (var tredje minut i faktisk tid). Därtill påverkas spelarens karma och kapital varje gång de besöker ett scenario och gör ett val inom scenariot. Även det fjärde målet anses vara uppfyllt inom ramen för ovan beskrivning av spelet. Det fjärde målet gick ut på att spelet ska ha ett engagerande och tydlig poäng- och belöningssystem. Vilket det har baserat på att karma och kapital påverkas av spelarens val.

Det femte målet specificerade att spelet skulle bestå av minst åtta spelaktiviteter, vilket det gör. Spelet innehåller åtta ”platser/scenarier” som spelaren kan besöka och varje ”plats” innehåller i sin tur 1-3 aktiviteter. Som spelet är utformat nu innehåller det totalt 29 spelaktiviteter.

Det sista målet var att spelet skulle vara satiriskt utmanande och utmana rådande samhällsstrukturer, vilket det gör i allra högsta grad.

All som allt har samtliga mål uppfyllts, men det finns samtidigt utrymme för förbättringar inom ramen för vissa av dem.

6 Slutsatser

Detta projekt har varit utmanande på många olika sätt. Den första utmaningen var att överhuvudtaget komma på en idé att genomföra och jag testade flertalet andra idéer innan jag bestämde mig för att gå vidare med spelet *Kapitalträsk.* I början av projektet lade jag ner en del tid på att läsa om hur koden generellt sätt är uppbyggd vid spelutveckling. Jag hittade en del grundläggande information och utgick från den i det fortsatta arbetet. Med facit i hand skulle jag dock ha anpassat klasserna i den generella strukturen i högre grad för att passa mitt flödesschema. På grund av framför allt tidsbrist kunde jag dock inte prioritera det då högsta prioritet har varit att överhuvudtaget få projektet klart.

Ett tydligt exempel är *Game()-klassen* som jag mycket hellre hade sett var uppdelad i flera mindre klasser. Som det ser ut nu innehåller *Game()-klassen* kod för både menyhanterings, spelflöde, tidshantering, kontroller av avkastning etcetera. Denna klass hade kunnat delas upp i flertalet mindre klasser så att *Game()-klassen* endast innehållit logiken för spelflödet.

Till exempel skulle jag vilja lyfta ut all tidshantering med *Timer* och *Stopwatch* till en egen klass som skulle kunnat få heta t. ex *TimeHandler.* Jag skulle också ha velat lyfta ut samtliga kontroller och hantering för avkastning till en egen klass som skulle kunnat heta t.ex *EarningsHandler.* Det finns alltså en del förbättringspotential vad gäller kodstrukturen. Då projektet var väldigt roligt att skapa hoppas jag finna tid till dessa förbättringar vid ett senare tillfälle. Ytterligare en spelkomponent som vore rolig att implementera i spelet är en lokal High Score-lista som lagrar spelarens/spelarnas snabbast vunna omgång.

En annan utmaning var att bygga upp ett logiskt spelflöde. Jag velade mellan att låta spelaren själv välja vilket scenario denne ville besöka eller att guida spelaren igenom spelet, men landade slutligen i att spelaren själv skulle få bestämma. Jag har emellertid ingen uppfattning kring om spelet ens är möjligt att klara, men hoppas att så är fallet.

Allt som allt har det varit ett väldigt lärorikt projekt där jag upplever att jag fått mer kunskaper i objektorienterad programmering, även om det också har kommit med en hel del frustration. Jag ser fram emot att vidareutveckla mina kunskaper i C#.

Källförteckning

[1] Gillis A. What Is Object-Oriented Programming (OOP)? [Internet]. Lewis S, editor. TechTarget. 2021. [cited 2024 Nov 28]. Available from: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

[2] Object-Oriented Programming | Brilliant Math & Science Wiki [Internet]. brilliant.org. [cited 2024 Nov 28]. Available from: https://brilliant.org/wiki/object-oriented-programming/

[3] Moore K, Njeru E, Pocevicius M. Classes (OOP) | Brilliant Math & Science Wiki [Internet]. Brilliant.org. 2016. [cited 2024 Nov 28]. Available from: https://brilliant.org/wiki/classes-oop/

[4] Bidragsgivare till Wikimedia-projekten. Objektorienterad programmering [Internet]. Wikipedia.org. Wikimedia Foundation, Inc.; 2003 [cited 2025 Nov 28]. Available from: https://sv.wikipedia.org/wiki/Objektorienterad_programmering

[5] Moore K. Methods (OOP) | Brilliant Math & Science Wiki [Internet]. Brilliant.org. 2019. [cited 2024 Dec 5]. Available from: https://brilliant.org/wiki/methods-oop/

[6] Static vs Instance Method [Internet]. Board Infinity. 2023. [cited 2024 Dec 5]. Available from: https://www.boardinfinity.com/blog/static-vs-instance-method/

[7] Doherty E. What is Object-Oriented Programming (OOP)? [Internet]. Educative. 2015 [cited 2024 Dec 5]. from: https://www.educative.io/blog/object-oriented-programming#Attributes

[8] Aggarwal A. Introduction to .NET Framework [Internet]. GeeksforGeeks. 2018. [cited 2024 Dec 5]. Available from: https://www.geeksforgeeks.org/introduction-to-net-framework/

[9] W3schools. Introduction to C# [Internet]. www.w3schools.com. [cited 2024 Dec 5]. Available from: https://www.w3schools.com/cs/cs_intro.php

[10] BillWagner. Main() and command-line arguments [Internet]. learn.microsoft.com. [cited 2024 Dec 5]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/main-command-line

[11] BillWagner. The C# type system [Internet]. learn.microsoft.com. [cited 2024 Dec 13]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/

[12] BillWagner. Organizing types in namespaces - C# [Internet]. Microsoft.com. 2024 [cited 2024 Dec 13]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces

[13] BillWagner. Classes [Internet]. learn.microsoft.com. [cited 2024 Dec 13]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes

[14] BillWagner. Egenskaper - C# [Internet]. Microsoft.com. 2024 [cited 2024 Dec 13]. Available from: https://learn.microsoft.com/sv-se/dotnet/csharp/programming-guide/classes-and-structs/properties

[15] Bill Wagner. Översikt över metoder - C# [Internet]. Microsoft.com. 2024 [cited 2024 Dec 13]. Available from: https://learn.microsoft.com/sv-se/dotnet/csharp/methods