Projektuppgift

Programmering i C#.NET

Projektuppgift

Andjela Jankovic

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: Förnamn Efternamn, xxxxx0000@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin, år: XX, 20XX

Innehållsförteckning

Projekt
Andjela Jankovic 2025-10-23

1 Introduktion

I dagens vardagsekonomi består många människors utgifter av små, återkommande belopp som ofta blir svåra att överblicka över tid. Det kan handla om matinköp, transportkostnader eller spontana köp som sker dagligen utan att man reflekterar över hur de påverkar den ekonomiska helheten. Utan rätt verktyg finns en risk att pengarna försvinner utan kontroll vilket gör budgetplanering svår och bidrar till bristande ekonomimedvetenhet.

Det problem som detta projekt riktar sig till är avsaknaden av ett enkelt och lättanvänt verktyg för att registrera och följa upp sina utgifter. Många befintliga lösningar kräver antingen konto registrering, molnkoppling eller är anpassade för mer avancerad ekonomihantering. Syftet med projektet är därför att utveckla en liten men funktionell applikation som ger användaren kontroll över sin privatekonomi utan krånglighet.

Projektet resulterar i ExpenseApp en konsolbaserad applikation utvecklad i C# med ramverket .NET. applikationen möjliggör registrering av utgifter med datum, kategori, belopp och beskrivning samt lagrar information lokalt i en JSON fil. Utöver detta erbjuder programmet funktioner för att filtrera utgifter baserat på exempelvis månad och kategori, ta bort registrerade poster, sammanställa rapporter samt exportera resultat till CSV format.

Applikationen är avgränsad till att hantera endast en användare åt gången och har ingen webb- eller databasuppkoppling. Fokus ligger istället på att demonstrera grundläggande programmeringsprinciper som objektorienterad programmering, filhantering och felhantering. Programmeringsstrukturen följer Single Responsibility Principle (SRP) vilket innebär att applikationen är uppdelad i separata klasser med tydliga ansvarsområden.

2 Teori

För att kunna förstå hur den utvecklade applikationen fungerar är det viktigt att beskriva de centrala tekniska begrepp och metoder som används. Detta kapitel behandlar de teoretiska grunderna som projektet bygger på med fokus på C#, objektorientering, .NET, filhantering med JSON, serialisering och felhantering.

2.1 C# som programmeringsspråk

C# är ett modernt, objektorienterad programmeringsspråk utvecklat av Microsoft. Språket används inom flera olika områden såsom webbaplikationer, spelutveckling, backend programmering och desktop program. C# är starkt typat vilket innebär att datatyper måste definieras tydligt. Detta minskar risken för logiska fel och gör programkod mer robust och underhållbar (1). I detta projekt används c# främst för att strukturera logiken i applikationen genom klasser, metoder och datahantering.

2.2 .NET platformen

.NET är en utvecklingsplatform som stödjer körning av c# program och tillhandahåller ett omfattande standardbibliotek. Plattformen innehåller färdiga funktioner för bland annat filhantering, datakonvertering, konsolutskrift, beräkningar och dataserialisering (2). Projektet bygger på .NET 8 SDK vilket verifierades genom kommandot dotnet –version i terminalen som gav resultat 8.0.414. Detta innebär att applikationen kan köras på flera operativsystem med hjälp av .NET runtime (Microsoft.NETCore.App 8.0).

2.3 Objektorienterad programmering (OOP)

Applikationen är byggd enligt objektorienterad programmering. OOP är ett arbetssätt där programstruktur baseras på objekt som representerar verkliga saker eller koncept. Dessa objekt skapas från klasser som beskriver vilka attribut (data) och metoder (funktioner) objektet ska ha (3). I ExpenseApp har till exempel en utgift en egen klass (Expense) med egenskaper som id, datum, kategori,

beskrivning och belopp. Ett viktigt princip inom OOP som applikationen följer är SRP. Den innebär att varje klass endast ska ha ett tydligt ansvar. I projektet innebär det:

KLASS	**ANSVAR**
Expense	Representerar en utgift
ExpenseRepository	Lagrar och hämtar data
ExpenseService	Affärslogik för utgifter
Reportservice	Skapar rapporter CSV
Menu	Hanterar konsolgränssnittet
Program	Startpunkt som kopplar ihop allt

2.4 Filhantering, JSON och serialisering

För att spara användarens utgifter mellan körningar används JSON, ett vanligt textbaserat format för lagring av data (4). Informationen sparas i en fil (Data/expenses.json) där varje utgift lagras som ett objekt. Applikationen använder serialisering för att konvertera listor av utgifter till JSON och deserialisering för att läsa tillbaka filen till c# objekt (5).

2.5 Felhantering

Felhantering är en viktig del av programutveckling. För att undvika att programmet kraschar används try-catch tillsammans med validering av inmatning. Exempelvis kan användaren inte skriva in tom kategori eller negativt belopp. Vid felmeddelanden får användaren instruktionen att försöka igen utan att programmet avslutas (6).

3 Metod

Arbetet med applikationen har genomförts med en praktisk och strukturerad utvecklingsmetod där lösningen delades upp i mindre delar för att möjliggöra stegvis implementering. Utvecklingen var baserad på en iterativ metod där en funktionalitet i taget byggdes, testades och förbättrades innan nästa steg kunde påbörjas. Detta arbetssätt valdes för att förenkla felsökning och säkerställa att programmets grundstruktur var stabil innan mer avancerade funktioner lades till.

För att lösningen skulle bli tydlig och enkel att underhålla användes en objektorienterad design med principen SRP som grund. Detta innebar att applikationen delades upp i separata klasser utifrån ansvar. Datamodellen definierades i klassen Expense som representerar en post av en utgift. Lagringslogik separerades i klassen ExpenseRepository som ansvarar för att spara och läsa utgifter från en JSON fil. Programmets logik hanterades av klassen ExpenseService där metoder utvecklades för att lägga till, ta bort, filtrera och summera utgifter. Användarinteraktionen placerades i klassen Menu som presenterar en textbaserad meny och tar emot inmatning från användaren. På detta sätt delades lösningen upp utifrån funktionalitet och varje del kunde utvecklas och testas oberoende av övriga komponenter.

Metoderna som användes för att lösa uppgiften bygger på grundläggande objektorienterad programmering där logiken delas in i klasser med tydliga ansvarsområden. Utgifterna lagras i en lista i minnet under programmers körning och sparas sedan permanent till en JSON fil. Filhanteringen sker genom serialisering med hjälp av System.Text.Json vilket gör det möjligt att omvandla listan av utgifter till textformat och tillbaka igen. Felhantering och validering har använts för att säkerställa att programmet inte kraschar vid felaktig inmatning. För att underlätta vidare analys av utgifter finns även möjlighet att exportera en rapport till CSV format vilket är ett enkelt textformat som kan öppnas i exempelvis Microsoft excel eller google sheets.

Utvecklingen utfördes med programmeringsspråket C# och ramverket .NET 8 SDK. Projektet skapades i Visual Studio Code och versionhanterades med Git. Projektet är även publicerat i ett Github repository för spårbarhet och redovisning.

Testningen utfördes löpande genom manuell interaktion i konsolen. Vid varje nytt moment verifierades att funktionaliteten utförde rätt beräkningar och att felaktig inmatning hanterades på ett kontrollerat sätt utan att programmet avbröts. Utvecklingsmetoden möjliggjorde både tydlig strukturering och enkel felsökning samtidigt som programmets funktionalitet byggdes upp stegvis utifrån kravspecifikationen.

4 Konstruktion

Det praktiska arbetet med applikationen inleddes med en planeringsfas där programmets struktur och funktionella krav definierades. Eftersom målet var att skapa en enkel men tydlig applikation delades projektet tidigt upp i separata delar. Detta underlättade både implementeringen och senare vidareutveckling. Applikationens struktur planerades enligt en lagerbaserad design med tre huvudområden: datalager, logiklager och användargränssnitt. Denna uppdelning säkerställer att logiken inte blandas med presentation och att varje del kan justeras vid behov.

4.1 Planering och design

Arbetet inleddes med att identifiera vilka data applikationen skulle hantera och hur dessa skulle representeras. Jag valde att använda en enkel datastruktur där varje utgift utgör en egen post med ett unikt ID. Därefter skapades en översiktlig modell av programmets flöde:

1. Användaren möts av en meny med val för att registrera eller visa data
2. Vid registrering kontrolleras att inmatad data är korrekt innan den sparas
3. Utgifter lagras i en lista i minnet under körning
4. Vid avslutning eller data lagring skrivs listan till en JSON fil
5. Vid start läses filen automatiskt in igen så att data inte går förlorad

4.2 Datamodell och informationshantering

Applikationens kärna utgörs av klassen Expense som beskriver en utgift med egenskaperna datum, kategori, beskrivning och belopp. För att data inte skulle blandas ihop mellan körningar valdes att spara dessa utgifter till fil. JSON format användes eftersom det är textbaserat, lättläst och funkar bra för mindre datamängder.

Vid programmets start laddas tidigare sparade utgifter automatiskt från fil med hjälp av serialisering. När användaren lägger till en ny utgift valideras data innan den registreras. Exempelvis tillåts inte negativa belopp eller tomma kategorier. Om användaren matar in ogiltiga värden skrivs ett felmeddelande ut istället för att programmet kraschar. Dessutom genereras varje nytt objekt automatiskt ett löpnummer för att underlätta identifiering vid borttagning.

4.3 Beräkningar och CSV

Beräkningar och rapportfunktioner hanteras genom klassen ExpenseService. För att summera utgifter per kategori och per månad används LINQ (Language Integrated Query) vilket är ett inbyggt verktyg i C# för att hantera och bearbeta data i listor på ett strukturerat sätt. LINQ gör det möjligt att filtrera, gruppera och beräkna värden på ett effektivt sätt utan att behöva skriva manuella loopar (7). För att summera utgifter per kategori grupperas alla utgiftsposter baserat på kategorifältet och sedan summeras varje grupps totalbelopp. Motsvarande metod används även för att sammanställa utgifter per månad genom att gruppera utgifterna efter formatet YYYY-MM. Dessa beräkningar utgör grunden för rapportfunktionen i appen. Ett kodutdrag som visar hur summeringen är implementerad kan ses i Figur 1. Metoderna SumByCategory() och SumByMonth() använder LINQ GroupBy och Sum funktioner för att sammanställa resultaten. De returnerade värdena visas i konsolgränssnittet som summerade rapporter och kan även exporteras till CSV format för vidare analys i t.ex. Excel.

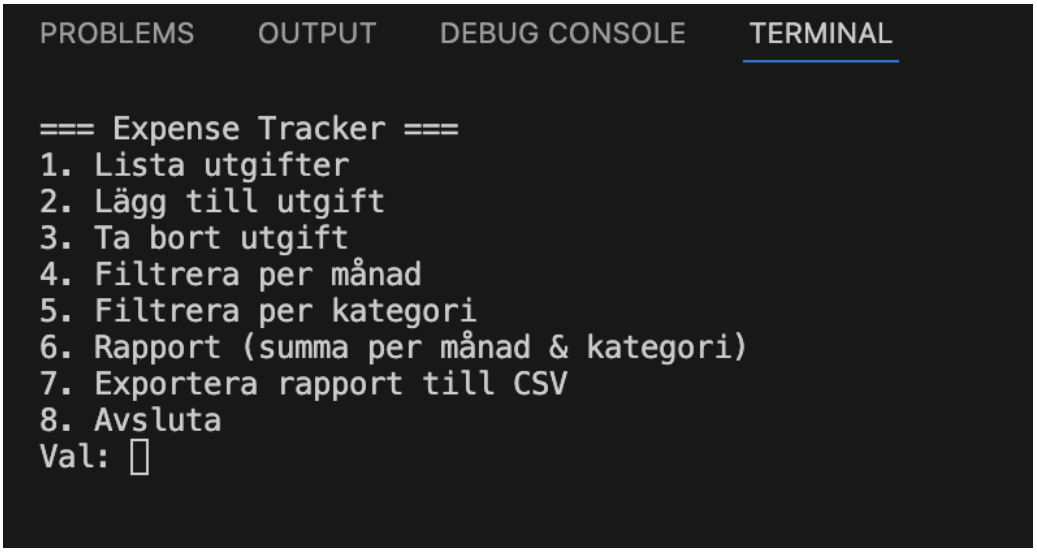
```
//summerar belopp per kategori
    2 references
    public IEnumerable<(string Category, decimal Sum)> SumByCategory(IEnumerable<Expense>? source = null)
        => (source ?? _items)
            .GroupBy(x => x.Category)
            .Select(g => (g.Key, g.Sum(x => x.Amount)));

//summerar belopp per månad över alla utgifter
    2 references
    public IEnumerable<(string YearMonth, decimal Sum)> SumByMonth()
        => _items
            .GroupBy(x => $"{x.Date:yyyy-MM}")
            .Select(g => (g.Key, g.Sum(x => x.Amount)))
            .OrderBy(x => x.Key);
```

Figur 1: Kodutdrag för summering med LINQ

4.4 Användargränssnitt

Menyn byggdes som en enkel loop i klassen Menu. Gränssnittet är textbaserat och användaren interagerar genom att göra val med hjälp av siffror. För tydlighet presenteras programmet med rubriker och konsekvent layout där varje tabellrad representerar en utgift. Efter varje operation pausas programmet så att användaren hinner ta del av resultatet.

Figur 2 visar en översikt av huvudmenyn i programmet.



Figur 2: Applikationens huvudmeny i konsolgränssnittet

4.5 Utvecklingsprocess

Konstruktionen av applikationen genomfördes stegvis. Utvecklingen började med att bygga den grundläggande strukturen och därefter lades funktionaliteten till i små iterationer. Grundfunktionerna utvecklades före filtrering och rapporter. Lagring i JSON och CSV export implementerades först när övriga funktioner fungerade korrekt. Samtidigt användes versionshantering genom Git vilket gjorde det möjligt att spara och dokumentera förändringar i projektet.

5 Resultat

Resultatet av arbetet blev en fullt fungerande konsolapplikation för hantering av utgifter. Applikationen uppfyller de krav som identifierades i problembeskrivningen och samtliga funktioner som planerades i metod- och konstruktionsdelen har implementerats. Programmet möjliggör inmatning av utgifter med datum, kategori, valfri beskrivning och belopp. Registrerade utgifter sparas automatiskt i en JSON fil vilket gör att de finns kvar även efter att programmet har avslutats. Användaren kan lista alla utgifter, filtrera dem per månad eller kategori och ta bort enskilda poster med hjälp av deras ID nummer. Utöver detta har programmet stöd för att sammanställa utgifter i en rapport som visar totalsumma per månad och per kategori samt exportera samma rapport till CSV format för vidare analys.

För att säkerställa att applikationen fungerar som avsett genomfördes löpande testning under utvecklingen. Testningen utfördes manuellt genom att köra programmet och mata in olika typer av data. Funktionerna för filtrering testades genom att lägga till utgifter med varierande datum och kategorier och kontrollera att rätt poster visades vid filtrering.
Felhantering testades genom att medvetet mata in ogiltiga värden såsom bokstäver istället för tal, negativt belopp eller tom inmatning. Programmet reagerade korrekt genom att ge tydliga felmeddelanden utan att avslutas oväntat eller krascha. Testerna visade även att programmet sparar data korrekt genom att utgifter fanns kvar vid nästa uppstart.

Rapportfunktionen testades genom att mata in flera utgifter och granska resultaten i både konsolutskriften och i den exporterade CSV filen. Summeringarna visade korrekt totalsummor baserat på datamängden. CSV filen kunde öppnas utan problem i Excel vilket bekräftar att exportformatet fungerar som avsett.

Sammanfattningsvis kan resultatet beskrivas som stabilt och funktionellt. De problem som identifierades i inledningen av projektet har adresserats genom att ge användaren ett enkelt och lokalt verktyg för ekonomisk översikt utan krångliga beroenden. Programmet uppfyller de mål som sattes upp för projektet och uppvisar en tydlig struktur som följer objektorienterade principer.

6 Slutsatser

Arbetet med applikationen genomfördes enligt plan och resulterade i en lösning som uppfyller samtliga krav. Utvecklingsprocessen visade dock att även en relativt enkelt applikation kan innehålla utmaningar som kräver både struktur och problemlösning.

Ett av de första problemen som uppstod var hanteringen av användarinmatning. Utan validering orsakade felaktiga värden som tomma fält eller ogiltiga tal oväntade programfel. Detta löstes genom att införa tydliga kontroller i gränssnittet vilket ökade programmets stabilitet. Ett annat område som krävde extra arbete var lagring av data. Först sparades utgifterna endast under programmets körning vilket innebar att allt försvann när programmet stängdes. Genom användning av JSON serialisering kunde problemet lösas på ett enkelt sätt utan att införa en databas.

Strukturmässigt fungerade uppdelningen mellan olika lager (modell, logik och användargränssnitt) mycket bra. Det gjorde programmet lätt att bygga ut och gav en tydlig överblick. Däremot hade det varit möjligt att ytterligare förbättra koden genom att införa mer enhetliga felmeddelanden eller ett separat lager för inputvalidering för att minska upprepning i koden. En annan förbättring hade varit att kommentera och dokumentera programmets metoder löpande under utvecklingen istället för i efterhand då detta hade sparat tid och förenklat arbetet.

Om applikationen skulle vidareutvecklats finns flera möjliga riktningar. Ett naturligt nästa steg vore att lägga till stöd för budgetmål eller statistik över längre perioder. Ett annat utvecklingsområde vore att byta ut JSON lagringen mot en databaslösning t.ex. SQLite för bättre skalbarhet. Det skulle även vara möjligt att bygga ut en webbversion av programmet samt införa flera användarkonton med inloggning.

Genom denna uppgift har förståelsen för objektorienterad programmering i C# stärkts, särskilt hur klasser och metoder kan struktureras på ett logiskt sätt. Arbetet gav även praktisk erfarenhet av filhantering, serialisering och enkelt rapportfunktionalitet. Dessutom visade projektet vikten av att arbeta iterativt och testa programmet kontinuerligt för att undvika större fel i utvecklingen. Sammantaget gav uppgiften både teknisk kunskap och insikter i strukturerad systemutveckling.

Källförteckning

[1] Microsoft Learn. A tour of C# language [Internet]. [uppdaterad 2025-01-07; citerad 2025-10-25]. Hämtad från: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview

[2] Microsoft Learn. What is .NET? [Internet]. [citerad 2025-10-25]. Hämtad från: https://learn.microsoft.com/en-us/training/modules/dotnet-introduction/2-what-is-dotnet

[3] MDN. Object-oriented programming [Internet]. [uppdaterad 2025-04-11; citerad 2025-10-26]. Hämtad från: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects/Object-oriented_programming

[4] MDN. Working with JSON [Internet]. [uppdaterad 2025-08-18; citerad 2025-10-26]. Hämtad från: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/JSON

[5] Microsoft Learn. JSON serialization and deserialization in .NET – overview [internet]. [uppdaterad 2025-01-29; citerad 2025-10-26]. Hämtad från: https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/overview

[6] Microsoft Learn. Exception Handling (C# Programming Guide) [Internet]. [uppdaterad 2023-03-14; citerad 2025-10-26]. Hämtad från: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/exception-handling

[7] Microsoft Learn. Language Intergrated Quiery (LINQ) [Internet]. [uppdaterad 2025-08-08; citerad 2025-10-26]. Hämtad från: https://learn.microsoft.com/en-us/dotnet/csharp/linq/