# Labbrapport: Labb 4

Författare: Felix Göstasson och Amanda Botros

Datum: 2026-01-09

Kursnamn: GIK299 – Objektorienterad Programmering

Examinator: Elin Ekman och Ulrika Artursson Wissa

## Innehåll

# 1. Introduktion

Denna rapport redogör genomförandet av laboration 4. Målet var att skapa ett system för att lagra persondata åt den fiktiva organisationen Angelic Good Guys. Uppgiften gick ut på att modellera verklighetsnära scenarier genom att bygga en konsolapplikation i C#. Syftet med labben är att fördjupa kunskaperna inom objekorienterad programmering genom att tillämpa egendefinierade typer såsom klasser, structs och enums.

Arbetet har genomförts i två delar (sprintar). Denna rapport beskriver hur programmet byggs upp steg för steg, från grundläggande datastrukturer till en interaktiv meny med listor och felhantering, samt redogör för de val och reflektioner som gjorts under processens gång.

# 2. Metod

## 2.1. Verktyg

- Microsoft Visual Studio Community 2022 (64-bit), version 17.14.20.
- .NET 9.0
- Versionshantering: Git och GitHub användes för att versionshanterna och arbeta tillsammans med koden. Arbetet strukturerades med branches (master och sprint-1) för att separera de olika delarna av laborationen.
- Kodexempel från Elin Ekmans (Ekman, 2023) GitHub-repon (Gik299_Ht23_Ref_L3, GIK299_ExceptionHandling och ResturantTables) användes som underlag för att förstå klasser, listor och felhantering.
- Microsoft Learn har använts som stöd för implementation av List<T> och felhantering med try-catch (Microsoft, u.å.)

## 2.2. Stegvis beskrivning av tillvägagångssätt

Arbetet delades upp i två sprintar för att strukturera och säkerställa att grunderna fungerade innan funktionaliteten togs fram.

### *2.2.1. Arbetsmetodik och fördelning*

Vi har arbetat enligt principer för parprogrammering där vi gemensamt diskuterat fram alla lösningar. Vi har suttit tillsammans (fysiskt eller via Zoom) och turats om att skriva koden, vilket gjorde att båda var delaktiga i hela processen.

För att strukturera arbetet tekniskt använde Git och GitHub. Genom att skapa olika ”branches” kunde vi versionshantera koden och tydligt separera den färdiga lösningen för Sprint 1 från vidareutvecklingen i Sprint 2.

### *2.2.2. Sprint 1: Struktur och grundläggande objekt*

I den första fasen låg fokus på att bygga upp programmets datastruktur. Vi började med att skapa de nödvändiga komponenterna: klass, struct och enum:

- Person.cs: Huvudklassen för att lagra information om personer.
- Hair.cs: En struct som vi skapade för att hantera hårdata.
- Gender.cs: Ett enum för att strikt definiera valbara kön
- Program.cs: Innehåller Main-metoden som är startpunkten för programmet, där vi instansierade och testade våra objekt.

För att verifiera strukturen instansierade vi ett hårdkodat person-objekt som skrevs ut i konsolen. Detta säkerställde att ToString()-metoden och kopplingarna mellan klasserna fungerade korrekt innan vi lade till användarinteraktion.

### *2.2.3. Sprint 2: Interaktion och lagring*

I nästa steg implementerade vi logiken för användarinteraktion i Program.cs. För att programmet skulle kunna hantera flera personer samtidigt bytte vi ut den enskilda variabeln mot en dynamisk lista.

En meny skapades för att låta användaren välja mellan att lägga till nya personer eller lista de som sparats. I samband med imatningen implementerades även felhantering för at säkerställa att programmet inte kraschar om användaren matar in data i fel format (exempelvis ett felaktigt datum). Avslutningsvis testkörde vi programmet genom att lägga till flera personer och kontrollera att de sparades i listan, för att vara säker på att allt fungerande som det skulle.

## 2.3. Förutsättningar för att göra labben

För att kunna kompilera och köra programmet krävs att ramverket .NET 9.0 är installerat på datorn. Koden är skriven i C#, vilket innebär att en utvecklingsmiljö (IDE) med stöd för detta språk krävs. Vi rekommenderar Microsoft Visual Studio 2022 med workloaden ”Desktop development with .NET” installerad. Utöver detta krävs inga särskilda inställningar.

## 2.4. Testning av koden

Vi har genomfört manuell testning löpande under laborationens gång för att säkerställa programmets logik fungerar som förväntat.

### *2.4.1 Funktionstester*

Vi verifierade huvudflödet genom att starta programmet, för att lägga till ett antal testpersoner via menyn och sedan välja alternativet för att lista dem. På så sätt kunde vi bekräfta att objekten lagrades korrekt i listan och att utskriften matchade den inmatade informationen.

### *2.4.2 Felhantering*

För att testa programmets robusthet genomförde vi medvetna försök att krascha det genom felaktig inmatning. Vi testade specifikt att mata in bokstäver eller ogiltiga format när programmet förväntade sig ett specifikt val eller datum. Det bekräftade att vår try-catch-blockering fångade felen och visade ett felmeddelande i stället för att avsluta programmet.

## 2.5. Etiska överväganden

Inga särskilda etiska överväganden har bedömts nödvändiga. Applikationen hanterar enbart fiktiva data som matas in manuellt vid testtillfället, och ingen information lagras permanent eller delas med tredje part. Därmed har inga verkliga personuppgifter eller annan känslig data behandlats under laborationens gång.

# 3. Resultat

Laborationen resulterade i en fungerande konsolapplikation som uppfyller samtliga krav i kravspecifikationen. Programmet kan skapa objekt av klassen Person, lagra dessa i en dynamisk lista och skriva ut dem korrekt formaterade i konsolen.

## *3.1. Koppling mellan testning och resultat*

Genom den manuella testningen kunde vi verifiera att flödet i applikationen är logiskt och att menyhanteringen fungerar felfritt. Under utvecklingen stötte vi på ett problem gällande inmatning av datum. Vi upptäckte att om användaren skrev in text eller ett felaktigt datumformat, så kraschade applikationen.

Lösningen blev att implementera en try-catch-loop inuti inmatningsmetoden. Resultat av detta syns i **Bild 1** där programmet fångar felet och ber användaren försöka igen istället för att avsluta.

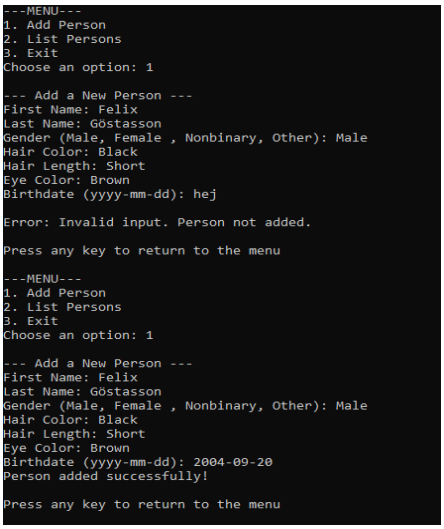


**Bild 1:** Skärmdump som visar hur programmet hanterar felaktig inmatning.

Efter att ha verifierat felhanteringen testade vi att lägga till flera personer för att se att listan sparade allt korrekt. Som **Bild 2** visar kunde vi lagra och skriva ut tre olika personer med korrekt data.

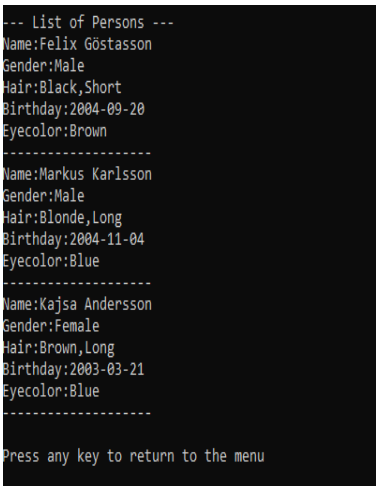


**Bild 2:** Skärmdump av programmets utskrift (”List Persons”) med tre sparade objekt.

# 4. Diskussion och reflektion

Arbetet med laborationen resulterade i en stabil applikation som utför sitt syfte. Den största tekniska utmaningen var hantera användarens inmatning på ett korrekt sätt. I början av utvecklingen kraschade programmet om användaren skrev in fel format för datum, vilket vi löste genom att implementera ett try-catch-block.

Vi har fått fördjupad förståelse för objektorientering. Genom att kapsla in data i objekt i stället för använda lösa variabler, blev koden betydligt mer lätthantering när vi skulle skala upp från en person till en lista. Vi utgick från kursens kodexempel (Ekman, 2023) för att strukturera upp korrekt struktur på klasser och properties.

Arbetsmetodiken parprogrammering visade sig vara mycket effektiv för att undvika logiska fel och hålla ett högt tempo. Genom att använda GitHub och ”Branches” kunde vi dessutom testa nya funktioner utan att riskera den fungerande koden.

Programmets största begränsning är att data endast sparas i arbetsminnet. En naturlig vidareutveckling vore att implementera något som gör att personlistan sparas permanent även efter programmet stängts av.

## 4.2. Diskussion kring resultat

I det stora hela fick vi ett förväntat resultat där applikationen uppfyller kravspecifikationen. Det som framför allt stack ut under arbetet var hur kritisk imatningsvalidering visade sig vara.

Vi märkte under testningen att DateTime.Parse var mycket känsligare än vi trott. Utan felhantering kraschade programmet direkt vid felaktigt format. Som vi redovisade **Bild 1** (se kapitel 3) löste vi detta genom att kapsla in i koden i ett try-catch-block.

Utöver detta fungerade while-loopen och menysystemet precis som förväntat. Resultatet av listan och utskrifterna kan ses i **Bild 2**.

## 4.3. Reflektion kring sprint 1

Under arbetet med Sprint 1 har vi fått praktisk övning i att bygga upp egna datatyper. Det blev tydligt hur enum är smidigt för att begränsa valmöjligheter (som för Gender, vilket minskar risken för felaktig data i jämförelse med en vanlig sträng. Vi funderade över skillnaden struct och class. Eftersom structen Hair bestå av enkla data (färg och längd) kändes en struct passande, medan klassen Person som är mer komplex fick vara en class. Att använda DateTime för födelsedagen var nyttigt för att se hur C# hanterar datumobjekt, vilket kommer underlätta om man senare vill räkna ut ålder. Inför Sprint 2 tar vi med oss vikten av att testa koden ofta. Att Person klassen fungerar redan nu gör att vi kan fokusera helt på logiken för listor och menyer i nästa moment.

## 4.4. Reflektion kring sprint 2

Under arbetet med Sprint 2 märkte vi en tydlig skillnad i arbetsbelastning. Sprint 2 krävde betydligt mer fokus på logiskt tänkande och flödeshantering. Baserat på erfarenheterna från den första sprinten valde vi strikt att arbeta med parprogrammering. Att vara två som resonerade kring kodens flöde minskade risken för logiska fel.

Vi hade inledningsvis räknat med att själva kodandet skulle ta mest tid, men insåg snabbt att felhantering krävde mer utrymme än förväntat. Versionshantering fick också avsevärt större vikt vilket gjorde att vi såg till att koden fungerade i vår utvecklingsbranchen innan vi mergade, för att inte förstöra den grund vi byggt i föregående sprint.

## 4.5. Reflektion kring alternativa lösningar

Om vi hade fått göra om laborationen hade vi framför allt velat införa en konstuktor i klassen Person. I dagsläget skapar vi ett tomt objekt och sätter värdena efteråt via properties, vilket innebär en risk att man glömmer ange viktiga data. En konstruktor hade tvingat oss att ange alla värden direkt vid skapandet. Detta hade garantera att ingen ”halvfärdig” person kan existera i systemet.

Vi hade även kunnat förbättra kodsturkturen genom att bryta ut all inmatningslogik från Program.cs till en separat klass. Genom att isolera användargränssnittet från huvudprogrammet hade vi fått en renare kodbas.

# Frågor till AI-verktyg

Verktyg: Google Gemini

Fråga/prompt: ”Jag behöver hjälp att formulera en diskussion kring alternativa lösningar (konstruktorer vs properties).”

På vilket sätt svaret användes: Svaret användes för att hjälpa till att förtydliga skillnaden mellan konstruktor jämfört med att sätta properties i efterhand, vilket användes som underlag för reflektionen i avsnitt 4.5.

Verktyg: Google Gemini

Fråga/prompt: ”Mitt program kraschar när användaren skriver in (t.ex. ”Hej”) istället för ett datum vid DateTime.Parse. Hur kan jag förhindra kraschen och i stället visa ett felmeddelande?

På vilket sätt svaret användes: AI-verktyget förklarade konceptet med try-catch-block på ett pedagogiskt sätt och gav ett kodexempel. Vi använde detta för att implementera felhanteringen.

# Referenser


Ekman, E. (2023). *Gik299_Ht23_Ref_L3* [Källkod]. GitHub. Hämtad 2025-12-19 från https://github.com/eemhda/Gik299_Ht23_Ref_L3

Ekman, E. (2023). *GIK299_ExceptionHandling* [Källkod]. GitHub. Hämtad 2025-12-22 från https://github.com/eemhda/GIK299_ExceptionHandling

Ekman, E. (2023). *RestaurantTables* [Källkod]. GitHub. Hämtad 2025-12-22 från https://github.com/eemhda/RestaurantTables

Microsoft. (2025). *Exception-handling statements*. Microsoft Learn. Hämtad 2025-12-22 från https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements

Microsoft. (u.å.). *List<T> Class*. Microsoft Learn. Hämtad 2025-12-23 från https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-8.0