PRAGUE PARKING 2

Avslutande inlämningsuppgift i kursen Arkitektur av applikationer i .NET C#

Prague Parking 2.0 – OOP-version

Bakgrund

Parkeringshuset i Prag behöver uppdateras och förbättras. Den är lite knölig att underhålla och stränghanteringen av data i arrayen har visat sig vara inflexibel när kunden säger att man i framtiden kommer att ta emot olika sorters fordon och kanske även olika stora parkeringsrutor.

Grundförutsättningarna är fortfarande desamma. Samma funktionalitet, endast parkering av MC och bilar i nuläget. Kunden har dock pratat om bussar och cyklar. Systemet skall registrera när fordonen kommer in och tala om hur länge de stått parkerade när de lämnas ut.

Kundens nya krav på systemet

- Data skall sparas så att det inte försvinner när programmet stängs av.
- Det skall gå att hantera P-platser av olika storlek så att även stora fordon (bussar) och små fordon (cyklar) kan hanteras.
- En karta över P-huset skall visas, så att man enkelt kan se beläggningen. Det skall gå att stå en bit ifrån skärmen och se översiktligt hur många platser som är lediga
- Det skall finnas en prislista i form av en textfil. Filen läses in vid programstart och kan vid behov läsas in på nytt via ett menyval. Filen skall gå att ändra även för en med låg IT-kunskap, så formatet behöver vara lätt att förstå. (TIPS: om allt efter ”#” på en rad är kommentarer, kan man ha dokumentation inne i själva filen)
- Det skall finnas en textfil med konfigurationsdata för systemet. Filformatet är fritt, men XML eller JSON kan vara lämpliga att använda.
- I konfigurationsfilen skall man kunna konfigurera
    - Antal P-platser i garaget
    - Fordonstyper (Bil och MC, men det kan komma fler)
    - Antal fordon per P-plats för respektive fordonstyp
- Prisstrukturen är till en början
    - Bil: 20 CZK per påbörjad timme
    - MC: 10 CZK per påbörjad timme
    - De första tio minuterna är gratis
- Filerna för prislista och konfiguration kan gärna kombineras till en fil
- Applikationen skall ha ett grafiskt användargränssnitt
    - WinForms eller ett webb-gränssnitt är acceptabla
    - Konsolapplikation är acceptabelt **om Spectre.Console** eller motsvarande används

Tekniska krav, nya i version 2.0

Alla fordon skall nu hanteras som objekt. Det skall finnas minst fem klasser:

1. Fordon

2. Bil, som ärver från Fordon
3. MC, som ärver från Fordon
4. ParkeringsPlats
5. ParkeringsHus

Om ni föredrar engelska namn på saker och ting:

1. Vehicle
2. Car, som ärver från Vehicle
3. MC, som ärver från Vehicle
4. ParkingSpot
5. ParkingGarage

Det kan mycket väl visa sig att det behövs fler klasser. Det kan också behövas en enumerator eller två.

P-huset kan med fördel innehålla en lista av P-platser. Fortfarande gäller att det finns 100 P-rutor som standardvärde, men det kan ändras med konfigurationsfilen.

Vidare skall lösningen innehålla separata projekt (dvs klassbibliotek) för

- Central logik
- Dataåtkomst
- Användargränssnitt

Inlämning

Inlämning sker i grupp. Grupperna skapas i samband med projektstart.

Inlämning görs i lärplattformen. Antingen görs en ZIP-fil av hela er solution i Visual Studio, eller så bifogar ni en länk till ert repo i GitHub.

Betygskrav

För G

- För betyget G skall alla kraven ovan vara uppfyllda.
- Uppgiften löses med den kunskap vi hittills lärt oss, såsom klasser med arv, kollektioner, loopar och listor
- Data om de parkerade fordonen lagras i en databas
    - Databasen fungerar samtidigt som testdata
    - Databasen skall uppdateras varje gång det sker en förändring av fordonen i P-huset (incheckning, utcheckning, förflyttning)
- Vid uppstart skall programmet läsa in inställningar för programmet från konfigurationsfilen
- Vid uppstart skall programmet läsa in data om parkerade fordon från databasen
- Inlämning skall ske via GitHub eller en uppladdad ZIP-fil
- Applikationen skall gå att köra på en dator annan än er egen (dvs på lärarens dator)
- Om det behövs några speciella handgrepp för att köra applikationen (utöver att trycks F5 eller Ctrl-F5 i Visual Studio) så skall dessa dokumenteras. Använd README.MD i GitHub för ändamålet.

Prague Parking 2.1 - För er som vill ha VG på uppgiften.

När ni är "klara", dvs har ett fungerande program med en meny, med Prague Parking 2.0:

Skapa ett nytt projekt i er Solution med lämpligt namn. Det skall givetvis framgå tydligt att det är version 2.1

Kopiera innehållet från 2.0-projektet till 2.1-projektet och fortsätt att arbeta därifrån.

Uppgiften löses med den kunskap vi hittills lärt oss, såsom klasser med arv, kollektioner, loopar och listor.

Dessutom ska delegater, lambdauttryck och LINQ användas (minst två av dessa tre).

Vidare skall det finnas minst två gränssnitt (interface) i systemet. Det kan till exempel vara IVehicle och IParkingSpot. Kodning skall primärt göras mot dessa gränssnitt och inte de konkreta klasserna.

Klassen **ParkeringsHus** skall vara implementerad som en ***Singleton***.

Lägg till nedanstående funktioner.

1. **P-huset skall kunna hantera andra fordonstyper**
    a. Bussar, som tar fyra gånger så mycket plats som en bil. Endast de första 50 platserna har tillräckligt högt i tak för att bussar ska kunna komma in.
    b. Cyklar, som tar hälften så mycket plats som en MC
    c. Om en cykel har storleken 1, så har
        i. MC 2
        ii. Bil 4
        iii. Buss 16
    d. Fordonstyper och deras storlekar sparas i konfigurationsfilen
    e. Timtaxan för cyklar är CZK 5/timma, för bussar är det CZK 80 per timma. (Priset är alltså relaterat till fordonsstorlek – det kan med fördel utnyttjas)
2. Programmet skall byggas så att databasen automatiskt skapas och fylls på med testdata, om den inte redan finns.
3. Programmet skall ha en funktion för att läsa in nya konfigurationsdata, som kan användas när som helst. Den behöver således vara synlig i menyn. Kontroll av de nya data skall göras så att man in te kan mata in ogiltiga värden. Exempelvis skall det inte gå att ta bort P-platser (dvs minska på antalet P-platser) om det står fordon parkerade på någon av dem. Den nya konfigurationen skall återspeglas i användargränssnittet.

Ytterligare krav för VG

För VG i betyg krävs det, förutom en fungerande version 2.1 av systemet enligt ovan, att ni skriver en personlig reflektion över projektet och kursen. Reflektionen kan lämpligen struktureras enligt nedan:

1. Sammanfattning
2. Hur vi/jag löste uppgiften

3. Utmaningar i uppgiften och hur de löstes
4. Metoder och modeller som använts för att lösa uppgiften
5. Hur du skulle lösa uppgiften nästa gång, givet vad du vet nu
6. Slutsats hemuppgift
7. Slutsats kurs

En sådan reflektion behöver inte bli särskilt omfattande, men ett par – tre sidor är normalt.

Tips

Det går att skapa riktigt snygga konsolapplikationer. Om man utnyttjar vad som faktiskt går att göra så kan man få till applikationer som inte skäms för sig. Här är några länkar till tips, verktyg och filmer som kan hjälpa er en bit på väg

- Awesome console apps in C#, en artikel om hur man kan göra snygga konsolapplikationer
  - Med ett kodexempel
- Spectre.Console, ett bibliotek som gör det lätt att bygga snygga konsolapplikationer
  - Dokumentationen är utmärkt
- Terminal.Gui - Terminal GUI toolkit for .NET är ett alternativ till Spectre.Console. (Kallas även gui.cs ibland). Den känns litet mer komplex och riktar sig mot gamla unix-hackare
- Om man vill hålla på med kod på låg nivå, bör man studera Console Class dokumentation hos Microsoft.