NACKADEMIN

Inlämningsuppgift 2: Grupparbete

Följande problem ska lösas tillsammans i ett grupparbete och dels redovisas genom Github och dels genom en kort presentation med demo av Problem 3 (spel).

Tanken med grupparbetet är främst att alla ska sitta tillsammans och hjälpas åt att bryta ned problembeskrivningarna till ett lösningförslag genom diskussioner, hjälpas åt om någon kör fast och så vidare.

Parprogrammera gärna! Sitt inte själv och koda allt. Även om gruppen bygger en lösning tillsammans är det ett krav att alla ska få vara med och göra en liten del av programmet på sin egen dator. Sitt tillsammans och programmera två eller tre personer. Den ena skriver och den andre tänker och kommer med förslag och sedan byter ni plats. Om ni inte sitter i skolan och jobbar kan ni använda skärmdelning via Slack, TeamViewer, Skype eller liknande program.

Det är tillåtet att titta på andras lösningar och skriva av kod men alla i gruppen ska ha lösningen på sin egen dator och alla i gruppen ska göra commit till Github på en del av lösningen i respektive problem.

För att bli godkänd ska koden uppfylla kraven samt vara lätt att förstå och läsa. Du ska följa de konventioner som finns för namngivning och formatering av koden på ett konsekvent sätt.

Betyg

Samtliga problem måste vara lösta enligt kraven. Det går bara att få G och IG på inlämningsuppgiften. Uppgiften är obligatorisk för att kunna få G på hela kursen.

Redovisning

Redovisning sker via Github. Återkoppling vad som måste åtgärdas för att få godkänt sker löpande via kommentarer i Github när ni commitar ny kod.

Sista dag för inlämning är **måndag 1 oktober** men jag rekommenderar att ni arbetar löpande med problemen under hela perioden.

Studenter som inte deltar i grupparbetet måste redovisa samtliga problem individuellt tillsammans med övriga inlämningsuppgift senast 8 oktober.

NACKADEMIN

Problem 1: Sudoku – Enkla Pussel

Vi ska bygga ett program som kan lösa Sudoku pussel.

Sudoku är ett logikbaserat, kombinatoriskt nummer-placerings pussel. Målet är att fylla ett 9 × 9 rutnät med siffror så att varje kolumn, varje rad och var och en av de nio 3 × 3 rutorna innehåller alla siffror från 1 till 9.

Den som skapar ett pussel ger dig en del av lösningen så att några av rutorna redan har siffror. Vanligtvis finns det tillräckligt med nummer från början för att det bara ska finnas en unik lösning.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

I första steget ska vi bygga ett program som bara kan lösa enkla pussel där man inte behöver gissa och prova flera alternativ.

Vad du ska lära dig

Ta ett djupt andetag. Jag vet att det här troligtvis är den svåraste utmaningen du fått. Om du kör fast, ta ett steg tillbaka och reflektera över hur du eller någon annan spelar Sudoku på papper så du inte "fastnar" i koden.

Ditt mål här är att lära dig hur man modellerar ett relativt komplext verkligt system - en person som löser Sudoku. Du lär dig vikten av väl inkapslad kod och att använda en del grundläggande objektorienterad design.

Målsättning

Modellering: Skriv ner substantiv och verb i spelet

Tänk noga på alla substantiv och verb i ett Sudoku-spel. Det finns den som skapade pusslet (skapare). Det finns den som löser pusslet (spelaren). Vad heter de viktigaste delarna av sudokubrädet? Hur interagerar spelaren respektive skapare med dem?

Ett datorprogram som löser Sudoku simulerar spelaren, vilket innebär att ju bättre du kan tänka dig in i spelaren roll, desto mer sannolikt förstår du hur du skriver en Sudoku-lösare. Du kommer att bli frestad att fokusera på brädet först - är det en slags matris, en sträng, något annat? - men gör det inte! Att förstå personen som spelar spelet är nyckeln; koden för att hantera sudokubrädet är en detalj.

Modellering: Strategier för människor

Lös ett verkligt Sudoku-pussel, tryckt på ett papper. Spela det tillsammans med någon annan. Var uppmärksam på dig själv och på varandra.

1. Vilka strategier använder du och varför?
2. Hur väljer du var du ska börja?
3. Hur vet du när du verkligen ska sätta ett tal i en cell?
4. Använde du samma anteckningar/noteringar på pappret när du spelar Sudoku som de andra i gruppen? Varför? Om inte, varför valde du annorlunda?
5. Undviker du några strategier eftersom de är för tråkiga eller kräver att du kommer ihåg för mycket?

Det är viktigt att se att de strategier som fungerar för oss (människor) ibland kan vara mycket svåra att göra på en dator och vice versa. Strategier vi undviker eftersom vi skulle behöva skriva för mycket, använda för många papper eller kom ihåg för mycket är ibland enkelt för en dator.

Modellering: Pseudokod för första steget

Kom ihåg att för det första steget ska vi bara bygga en lösare som fyller i "logiskt självklara" rutor och som inte kräver någon gissning. Det löser inte varje Sudoku-bräde men det löser ofta de enklaste. Hur kan du avgöra när du fyllt i alla "logiskt självklara" rutor?

Skriv pseudokoden för det här steget själv och jämföra sedan den med varandra. Hur skiljer deb sig? Vilket tillvägagångssätt verkar mer vettiga? Hittar du några centrala metoder du behöver göra?

Till exempel, givet en cell/ruta, behöver du förmodligen minst tre metoder:

- Ge mig de andra cellerna i den cellens rad.
- Ge mig de andra cellerna i den cellens kolumn.
- Ge mig de andra cellerna i cellens ruta.

Nu är det dags för C#!

Din Sudoku-lösare ska ta en sträng så här som sin indata:
`619030040270061008000047621486302079000014580031009060005720806320106057160400030`

Varje på varandra följande 9 siffror representerar en rad av Sudoku-brädet och ’0’ representerar en tom cell. Det skulle fungera så här:

```
static void Main(string[] args)
{
    Sudoku game = new Sudoku("003020600900305001001806400"+
                             "008102900700000008006708200"+
                             "002609500800203009005010300");
    game.Solve();
    Console.WriteLine(game.BoardAsText);
}
```

Detta skulle skriva ut något som ser ut så här:

```
---------------------
4 8 3 | 9 2 1 | 6 5 7
9 6 7 | 3 4 5 | 8 2 1
2 5 1 | 8 7 6 | 4 9 3
---------------------
5 4 8 | 1 3 2 | 9 7 6
7 2 9 | 5 6 4 | 1 3 8
1 3 6 | 7 9 8 | 2 4 5
---------------------
3 7 2 | 6 8 9 | 5 1 4
8 1 4 | 2 5 3 | 7 6 9
6 9 5 | 4 1 7 | 3 8 2
---------------------
```

OBS! Den här första versionen kanske inte löser alla Sudoku-bräden. Det betyder att Solve() måste sluta när det inte längre kan hitta några ”logiskt självklara” alternativ och "ge upp". Vi nästa problem ska vi skapa den fullständiga versionen som också klarar av att gissa.

Utseendet är inte det viktiga utan det är att få till logiken runt att lösa/gissa på rätt sätt som du ska fokusera på.

Prestanda?

Oroa dig inte om prestanda än! Optimeringar kan man göra senare. Ren, logisk kod är viktigare och blir lättare att anpassa och göra om (refactor).

Problem 2: Sudoku – Svåra Pussel

Nu har ni gjort ett program som kan lösa enkla pussel. Nästa steg är att försöka få det att lösa svåra pussel. För att göra det är programmet tvungen att ”gissa” vilket tal som ska vara i en cell och göra om ifall det blev fel.

När ni löser det första problemet borde ni så småningom hittat en ganska enkel algoritm som passar bra för en dator som inte blir uttråkad.

```
Upprepa tills klar
    För varje cell i brädet
        Om cellen är tom
            Hitta alla tal som är möjliga i cellen
                Om endast ett möjligt tal för cellen
                    Sätt cell till det talet
                    Sätt flagga ej klar för att upprepa och gå igenom brädet igen
```

Om er lösning är mycket mer komplicerad bör ni överväga på att förenkla koden innan ni fortsätter med problem 2. Som en jämförelse är min Sudoku klass för att lösa problem 1 cirka 140 rader kod.

Modellering: Strategi för människor

Att gissa är naturligtvis inget en dator kan göra och tittar du på hur en människa löser Sudoku så ser du snabbt att det finns strategier för att hitta vilka tal man ska prova med för att hitta en lösning.

Lös ett medelsvårt Sudoku pussel på papper tillsammans. Var uppmärksam på dig själv och på varandra.

- Hur gör du när det inte finns något logiskt självklara tal att skriva i någon cell?
- När du behöver ”gissa” hur hittar du vilka tal som är möjliga alternativ?
- Hur gör du om du gissar fel?
- Hur upptäcker du att du gissat fel och måste prova med ett annat tal?

För människor finns många olika strategier för att med uteslutningsmetoden kunna hitta vilket tal som ska skrivas in i en cell när det finns fler än ett tillåtet alternativ. Att göra om alla dessa strategier till kod kan bli väldigt komplicerat.

Här är vi tvungna att tänka på skillnaden mellan människor och datorer. Människor kan göra små anteckningar och söka mönster enkelt medan datorer enkelt kan kopiera hela brädet och försöka med alla tillåtna tal i en cell i tur och ordning utan att tröttna.

Kan vi hitta en strategi som löser alla sudoku-pussel som passar en dator?

Modellering: Psuedokod för andra steget

För att lösa alla sudoku-pussel behöver vi lägga till ett nytt steg i programmet som tar vid när alla celler med logiskt självklara värden är ifyllda.

När programmet gått igenom brädet gång på gång och fyllt i alla celler där det bara finns ett tillåtet tal är antingen pusslet löst eller så finns det tomma celler kvar. Dessa tomma celler har fler än ett tillåtet tal.

För att lösa pusslet behöver vi hitta en tom cell och hämta alla dess tillåtna tal, sedan provar vi ett tillåtet tal i taget. Om du hittar tomma celler med inga tillåtna tal vet du att talet du talet du prövade inte fungerar och du får fortsätta och prova med nästa tal.

Skriv pseudokoden för det här steget själv och jämföra sedan den med varandra. Hur skiljer den sig? Vilket tillvägagångssätt verkar mer vettiga? Hittar du några centrala metoder du behöver göra?

Dag för C# kod!

Först när ni har hittat en algoritm och skrivit psuedokod är det dags att fortsätta skriva kod. Här är ett medelsvårt pussel.

```
static void Main(string[] args)
{
    Sudoku game = new Sudoku("037060000205000800006908000"+
                             "000600024001503600650009000"+
                             "000302700009000402000050360");
    game.Solve();
    Console.WriteLine(game.BoardAsText);
}
```

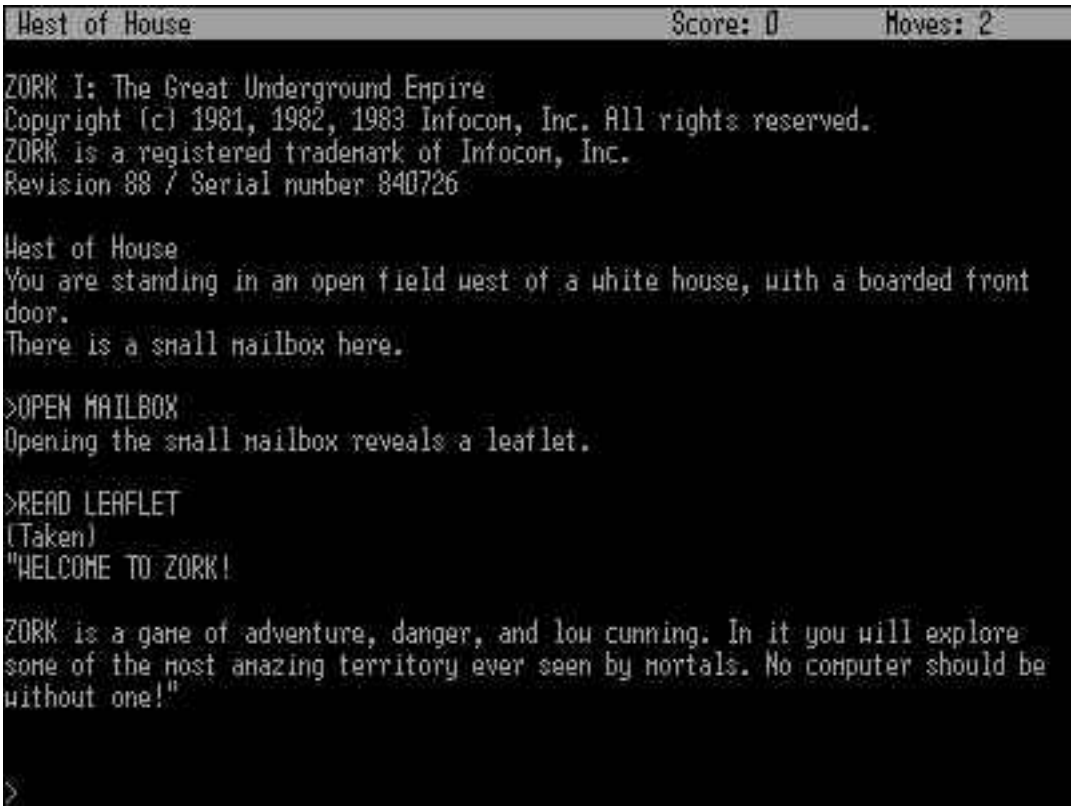
Prestanda?

Oroa dig inte om prestanda än! Optimeringar kan man göra senare. Ren, logisk kod är viktigare och blir lättare att anpassa och göra om (refactor). Det innebär att en del svåra Sudoku pussel kan ta väldigt lång tid att lösa.

Problem 3: Gör ett spel!

För länge länge sedan fanns det en typ av dataspel som kallades för "Text Adventures". I dessa spel skriver användaren in text-kommandon för att kontrollera en karaktär och påverka omgivningen. Målet med spelet är att användaren ger dessa kommandon för att förflytta sig genom spelvärlden för att uppnå ett förutbestämt mål och vinna spelet.

Nedan visas en utskrift från ett sådant spel (Zork I).



Som synes skriver spelaren in olika kommandon, som sedan tolkas av programmet. Om "rätt" kommandon skrivs in, svarar programmet med att utföra kommandot och förändra spelvärlden.

Er uppgift är att bygga ett enkelt ett Text Adventurespel med en egen unik handling. Spelet ska ha minst ett ”pussel” att lösa och det avslutas antingen av att du dör eller lyckas lösa alla pussel. Du ska kunna gå runt i spelet, ta upp föremål och använda föremålen för att lösa pussel.

Programmet ska ha följande egenskaper och funktioner:

Det ska vara en Console-applikation och användaren ska kommunicera med spelet via kommandon. Dessa kommandon ska vara text som användaren skriver in.

Programmet ska göra det möjligt att bygga upp en spelvärld (game) med följande funktionalitet:

- Spelkaraktär (player), detta är en representation av en person, som användaren kan flytta runt i spelvärlden mellan platser. En spelkaraktär har ett namn (fråga användaren vid spelets start) samt en lista med föremål som karaktären just nu bär på (inventory).

- Platser (room), som spelkaraktären kan flyttas mellan. Varje plats ska kunna ha en eller flera utgångar. En plats ska kunna markeras som "End Point" för spelet, dvs om man når dit har man klarat spelet.

- Utgångar (exit) kan antingen vara öppna eller låsta. En låst utgång ska kunna öppnas genom att använda ett visst föremål på utgången (t ex ge kommandot "Use key on door" för att öppna en dörr eller ”Use stone on window” för att krossa ett fönster).

- Platser kan innehålla föremål. Föremål ska kunna plockas upp från en plats, varvid den istället placeras på spelkaraktären (inventory). Det ska även gå att släppa föremål som spelkaraktären innehar, dessa hamnar då på den plats som spelkaraktären just nu befinner sig.

- Ett föremål ska kunna användas på ett annat föremål, för att ge spelkaraktären ett nytt föremål. T ex, om användaren skriver "Use corkscrew on bottle" och spelkaraktären innehar föremålen "corkscrew" och "bottle", så tas föremålet "bottle" bort och ersättas med föremålet "opened bottle".

De typer av kommandon som spelet ska kunna hantera är:

- Navigering mellan rum (t ex "go east"). Varje gång spelkaraktären flyttas in i ett rum, ska rummets beskrivning vis

- Ta upp och släppa föremål ("get key", "drop key"). Föremålet ska då flyttas från rummet till spelkaraktären eller tvärtom.

- Använda föremål på utgångar ("use key on door"), för att låsa upp utgångarna.

- Använda föremål på andra föremål ("use corkscrew on bottle"), för att byta ut det första föremålet mot något annat föremål.

- Visa rumsbeskrivning igen, tillsammans med listning över alla utgångar och föremål i rummet ("Look").

- Visa detaljerad information om ett föremål eller en utgång. Båda dessa ska kunna ha en längre textbeskrivning kopplade till sig, genom att skriva t ex "inspect door" så visas denna text.

Programmet ska meddela användaren om den inte förstår ett visst kommando, eller om kommandot inte går att utföra (t ex försöka gå till ett annat rum via en låst dörr). Meddelanden ska vara begripliga för användaren.

Programmet MÅSTE skrivas objektorienterat!

Det ska alltså finnas klasser för de olika typer av saker som finns i spelvärlden, klasserna har metoder och egenskaper och interagerar med varandra. Det ska sedan vara möjligt att skapa instanser av dessa klasser som "sätter upp" en spelvärld.

Koden skall fungera och applikationen skall gå att köra utan fel.