Inlämningsuppgift – Sudoku

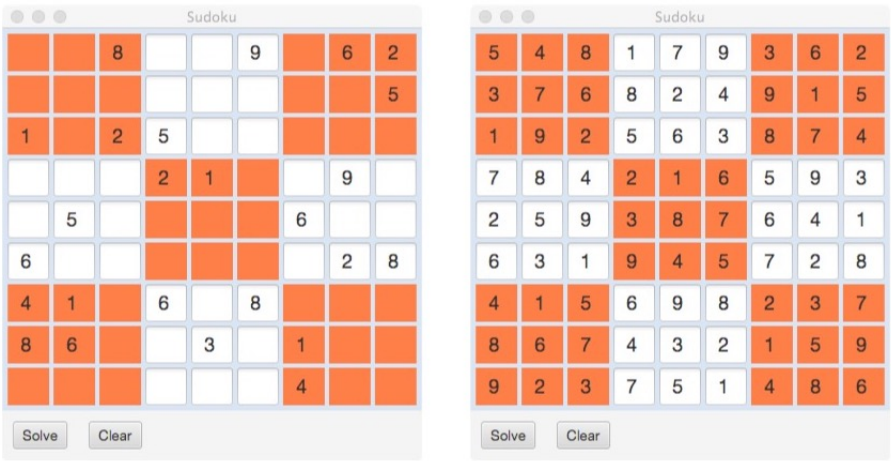
Uppgiften går ut på att skriva ett program som löser sudoku. Användare skriver in siffror i några rutor och kan sedan begära att få en lösning. Lösningen ska beräknas med rekursiv teknik med s.k. backtracking. Programmet ska ha ett grafiskt användargränssnitt.

Här nedan beskrivs vilken lösningsmetod som ska användas (avsnitt 1) och vilka klasser och interface som ska ingå (avsnitt 2).

1 Problemet

Sudoku är ett ”pussel” som består av 9*9 rutor. Några rutor är från början ifyllda med siffror mellan 1 och 9. Uppgiften går ut på att hitta en lösning som uppfyller följande krav:

- Varje ruta är fylld med en siffra mellan 1 och 9.
- Varje rad, vågrät och lodrät, innehåller siffrorna 1–9.
- Varje ”region” innehåller också siffrorna 1–9. Med region avses här en grupp av nio rutor motsvarande de grupper som färgats ljusa respektive mörka i figur 1.
- Ingen av siffrorna förekommer mer än en gång per rad, kolumn eller region.



Figur 1: Sudoku med ett antal från början ifyllda rutor samt lösning.

1.1 Lösning av sudokut

Ett sudoku representeras lämpligen av en klass som har en heltalsmatris motsvarande sudokuns rutnät som attribut.

Den viktigaste metoden i klassen är den som löser problemet. Utgående från ett delvis fyllt rutnät ska denna metod hitta en lösning eller meddela att ingen lösning finns. Problemet ska lösas rekursivt med s.k. backtrackingteknik.

1.2 Backtracking

Backtracking kan användas för problem där man genom att systematiskt prova ett begränsat antal möjligheter kan konstruera en lösning. En rekursiv lösning av sudokut bygger på följande resonemang: när man befinner sig i en viss ruta i kan man prova att fylla i siffran 1. Om det då går att lösa resten av sudokut så har man funnit en lösning, i annat fall får man prova siffran 2 etc. Man provar alltså i tur och ordning att fylla i 1 till 9 och gör rekursiva anrop som undersöker om det finns en lösning utgående från nästa ruta. Om en lösning finns är vi klara, annars måste vi backa tillbaks till den ruta vi närmast kom ifrån. Om det finns fler siffror att prova i denna ruta väljer vi nästa siffra i tur. Annars backar vi en ruta till etc. Denna backning (backtracking) sköts automatiskt av rekursionen genom att vi återvänder till anropande upplaga av den rekursiva metoden. På detta sätt kommer alla alternativ att systematiskt provas tills man hittar ett som lyckas eller kan konstatera att lösning saknas.

1.3 Rekursiv lösning för Sudoku

När användare matat in siffror i rutor och begärt att få se en lösning kan olika fall inträffa:

- Det saknas lösning. Då ska programmet upptäcka och meddela detta.
- Det finns flera lösningar. Då räcker det att programmet finner en lösning och presenterar denna.
- Det finns en enda lösning. Då ska denna hittas och presenteras.

Alla dessa fall kan hanteras av en rekursiv metod som använder backtracking. Metoden ska ha parametrar för rad och kolonn och t.ex. ha följande signatur:

```
private boolean solve(int r, int c);
```

För att lösa problemet startar man på rutan längst uppe till vänster, (0,0), vilket motsvarar ett anrop `solve(0,0)`. För en enskild ruta (nedan kallad aktuell ruta) finns det två fall:

1. Aktuell ruta är inte från början fylld (av användaren). Då provar man i tur och ordning att fylla den med något av talen 1..9. För varje sådant val kontrollerar man först att det är möjligt med hänsyn till reglerna för Sudoku. Om det är möjligt fyller man i rutan och gör ett rekursivt anrop för nästa ruta. Med nästa ruta avses här rutan till höger om aktuell ruta eller (om aktuell ruta är den sista på en rad) första rutan på nästa rad. Om det inte går att fylla aktuell ruta med något av alternativen eller om de rekursiva anropen returnerar `false` för alla de alternativ man provar, så markeras aktuell ruta som som ej ifylld (backtracking) och man returnerar `false`. Om däremot något av de rekursiva anropen returnerar `true` så har man hittat en lösning och kan returnera `true`.

2. Aktuell ruta är från början fylld (av användaren). Då ska vi inte prova några alternativ utan bara kontrollera att det som är ifyllt är ok enligt reglerna. Om så är fallet görs ett rekursivt anrop för nästa ruta och resultatet av detta returneras. Om den ifyllda rutan däremot inte uppfyller villkoren har man misslyckats och returnerar `false`.

För båda alternativen ovan gäller att om ”nästa ruta” inte finns (d.v.s. vi har gått igenom hela rutnätet) så har vi lyckats fylla i alla rutor och kan returnera `true`.

Observera att den här lösningen medför att många alternativ ska provas. I vissa fall av olösliga sudokun tar det mycket lång tid innan man får besked om att en lösning inte finns.

2 Deluppgifter

Programmet består av två delar. En del som utgör en modell av sudokun (sudokulösaren) och en del som utgör ett grafiskt användargränssnitt. Sudokulösaren ska vara helt oberoende av det grafiska användargränssnittet.

Interfacet SudokuSolver Det finns ett interface, `SudokuSolver`, som ska utgöra gränssnittet mellan sudokulösaren och det grafiska användargränssnittet.

Javadoc Metoderna i interfacet `SudokuSolver` ska förses med dokumentationskommentarer (javadoc). Vissa kommentarer är redan klara, men lägg till kommentarer där det saknas. Felaktig indata ska behandlas på motsvarande sätt som i de metoder med färdiga kommentarer.

Prova gärna att skapa javadoc-filer och kontrollera att dokumentationen ser bra ut.

Sudokulösaren Skriv en klass som beskriver sudokut och dess lösning enligt beskrivningen i avsnitt 1. Klassen ska implementera interfacet `SudokuSolver`.

Automatisk testning Skriv testfall med JUnit för metoderna i din sudokuklass. Bland testmetoderna ska ingå lösning av ett tomt sudoku, lösning av sudokut i figur 1 samt försök att lösa lämpliga fall av olösliga sudokun. Alla publika metoder ska ha anropats åtminstone en gång i testklassen.

Interfacet `SudokuSolver` ska användas som typ för sudokulösaren i testklassen. Det är bara när sudokulösarobjektet skapas som du ska använda ditt eget klassnamn.

Grafiskt användargränssnitt Programmet ska ha ett grafiskt användargränssnitt. Se figur 1 som exempel på hur det kan se. När programmet startar ska ett tomt rutnät med 9*9 rutor samt två knappar visas.

Rutorna kan vara textfält med plats för ett tecken vardera. Dessa textfält placeras på en panel. För att tydligt markera olika regioner kan olika bakgrundsfärg användas för textfälten.

Användare skriver in siffror i ett antal rutor. Därefter kan man be att få se en lösning genom att trycka på knappen ”Solve”. Knappen ”Clear” ska ta bort alla siffror från alla rutor.

I den lyssnaren som kopplas till knappen ”Solve” ser man till att läsa av alla textfälten och föra över motsvarande värden till sudokulösaren. Därefter anropas sudokulösarens metod för att lösa sudokut. Om en lösning finns hämtar man alla rutornas värden från sudokulösaren och visar dessa i motsvarande textfält. Annars visas ett dialogfönster där det anges att ingen lösning finns. Lyssnaren kopplad till knappen ”Clear” tömmer siffrorna i sudokut.

Programmet ska vara bekvämt att använda. Det ska inte krascha om användaren skriver in konstiga värden i rutorna. Om sudokut inte går att lösa ska användaren få chansen att ta bort en eller flera siffror och försöka igen. Det är alltså bara när användaren klickar på "Clear-knappen som hela sudokut ska tömmas.

Inuti koden för användargränssnittet ska Interfacet `SudokuSolver` användas som typ för sudokulösaren. Det är bara när sudokulösarobjektet skapas som du ska använda ditt eget klassnamn.