Labyrint-spillet

Regulær eksamensopgave, OOPD 2009/2010

Dirk Hasselbalch

22. januar 2010

1 Resumé

Du skal designe, implementere og teste dele af et simpelt computerspil. Nogle dele af programmet har du fået udleveret, og nogle dele skal du selv lave. Du vil også blive bedt om at ændre i noget af den udleverede kode.

Hver opgave består af en indledende beskrivelse efterfulgt af en række delopgaver. Det er kun i delopgaverne, du bliver bedt om at lave noget.

Dokumentationen til den udleverede kode findes i projektets `doc`-mappe, og kan som sædvanligt genereres vha. JavaDoc. Inden du starter med opgaven, rådes du til at danne dig et overblik over det udleverede programmel ved at orientere dig i dokumentation og kode.

2 Retningslinjer og formalia

Denne opgave skal løses individuelt. Der tillades ikke genaflevering. Under helt særlige omstændigheder (sygdom, dødsfald, m.v.) kan der gives en udsættelse efter dispensation fra studienævnet. En dispensationsansøgning skal være studienævnet i hænde inden afleveringsfristens udløb. Yderligere oplysninger kan findes i eksamensbestemmelserne for Det Naturvidenskabelige Fakultet samt studieordningen for Datalogiuddannelsen. Eksamenssnyd (herunder afskrift fra andre personer) behandles efter Københavns Universitets disciplinærbestemmelser og kan medføre annullering af samtlige eksamensresultater for alle kurser det pågældende studieår samt relegering. Eksamensopgaven udgøres af nærværende dokument samt eventuelle supplerende beskeder fra kursets undervisere, som vil blive gjort tilgængelige på kursusforummet på dikutal.dk [1]. Det er deltagerens ansvar løbende at holde sig orienteret på hjemmesiden.

2.1 Afleveringsformat

Der skal afleveres en jar-fil på OOPD-hjemmesiden [2] under punktet "Eksamen 2010" > "Eksamensopgaven 2010" med navnet

<efternavn>.<fornavn>`.eksamen.jar`

indeholdende programmet, dvs. alle Java-kildekodefiler, dokumentation, JUnit tests, og besvarelsesteksten i PDF- eller tekstformat. Mere specifikt skal jar-filen indeholde:

1. **Et dokument med navnet** `besvarelse.pdf` **eller** `besvarelse.txt` indholdende besvarelserne på opgaverne 1.3.1, 1.3.2, 2.1.2 og 2.1.4, inklusive evt. figurer. Alle førnævnte opgaver, der kræver tekstuel besvarelse i besvarelsesdokumentet er markeret med stjerne (*).

2. **Java-kildekode** forsynet med *kommentarer* – både JavaDoc og ikke-JavaDoc. JavaDoc-kommentarerne skal for hver udviklet klasse og grænseflade, hvor der ikke i forvejen er kommentarer, rumme en beskrivelse samt eventuelle pre- og postconditions. Ikke-JavaDoc-kommentarer forventes steder i selve kildeteksten, hvor det ellers kan være vanskeligt at forstå koden.

Det er vigtigt at du sikrer, at din kildekode kan importeres og kompileres af andre. Husk at aflevere **alt** kildekoden til programmet, ikke kun de dele, du selv har kodet eller udvidet.

3. **JUnit testklasser**, som krævet i opgave 2.1.3 og forsynet med JavaDoc-kommentarer. Alle JUnit tests skal ligge i pakken `test`.

4. **Dokumentation** i form af en `doc`-mappe indeholdende den fra JavaDoc genererede dokumentation i html-format.

Du bedømmes ud fra det, du har afleveret! Hvis du ved en fejltagelse ikke har inkluderet fx din kildekode, kan denne ikke bedømmes, og karakteren bliver derefter! Derfor:

> **Husk at teste din jar-fil, og tjek at al kildekode, dokumenter, kommentarer, javadoc dokumentation, figurer etc. er inkluderet!**

2.2 Arbejdsfordeling og bedømmelseskriterier

Nogle opgaver er mere omfangsrige end andre. Fordelingen af arbejdsbyrden på opgaverne er normeret til:

Opgave 1: Strategy Pattern Ca. 2 timer, eller ca. 10%.

Opgave 2: Lees Algoritme Ca. 8 timer, eller ca. 40%.

Opgave 3: Konfigurationsindlæsning Ca. 4 timer, eller ca. 20%.

Opgave 4: Swing og Model-View-Control Ca. 6 timer, eller ca. 30%.

Den endelige karakter tager udgangspunkt i besvarelsen af de enkelte delopgaver og hvor godt besvarelsen som helhed opfylder læringsmålene, som kan ses på kursets SIS-hjemmeside [3]. Det er tilrådeligt selv at forholde sig til læringsmålene når opgaven besvares. Det bemærkes at vægtningen af karakteren generelt vil afvige en smule fra vægtningen af den normerede arbejdsbyrde, samt at karakteren også påvirkes af hvorvidt det færdige program fungerer, og hvor godt det fungerer.

Der lægges vægt på at der afleveres et kørende program, dvs. at et simpelt og lille men veldesignet, kørende program foretrækkes frem for et ambitiøst eller stort program, der ikke rigtig virker. Det er vigtigt, at individuelle klasser er veldesignet, specificeret, implementeret og evt. uformelt afprøvet (hvis ikke andet er specificeret i opgaveteksten), selv hvis programmet som helhed ikke kan køre. Det anbefales stærkt at designe, implementere og afprøve et system med meget enkel funktionalitet, inden eventuelle udvidelser føjes til.

2.3 Tilladt samarbejde under eksamensugen

Bemærk følgende politik for samarbejde under eksamensugen:

1. Eksaminanderne må stille spørgsmål til hinanden, som er uafhængig af opgaven, f.eks. om de obligatoriske opgaver eller vedr. stof i en af lærebøgerne, Java API, vejledningerne eller andre almene ressourcer.

   - Tænk på denne eksamen som en "open book" eksamen: det er tilladt at bruge medeksaminander, lige som man bruger en bog eller anden alment tilgængelig ressource (som altså ikke ved noget specifikt om denne eksamensopgave).
   - Det er altså tilladt at at bruge hinanden som kilde for f.eks. hvordan Observer/Observable fungerer, hvor man kan finde noget læsbart om Swing JButtons, hvad der adskiller et `HashSet` fra et `TreeSet` m.m.
   - Det er *ikke* tilladt at spørge hinanden eller tredjepart, hvordan en anden har tænkt sig at løse eksamensopgaven.

2. Hver linie kode og tekst, der afleveres skal være produceret af eksaminanden og af eksaminanden alene.

3. Alle kan stille spørgsmål om opgaven til instruktorvagterne eller på diskussionsforum. Generelle svar (uden specifik kode eller specifikke designs for opgaven) på disse, herunder af andre studerende, er tilladt, men skal i denne forbindelse gives på diskussionsforummet [1] for at tilgå alle eksamensdeltagere og således ligestille alle indbyrdes.

4. Alle eksaminander er underlagt KUs eksamensregler i hele perioden fra fredag, kl. 14, til efterfølgende fredag. Det betyder, at snyd, forsøg på snyd og hjælp til at begå snyd (f.eks. ved at hjælpe nogen konkret med løsningen) resulterer i bortvisning fra eksamen samt anmeldelse til studielederen mhp. eventuel forelæggelse hos dekanen mhp. yderligere skridt (f.eks. bortvisning fra alle eksaminer i indeværende eksamensperiode). Det gælder for så vidt også ikke-OOPD/studerende, som hjælper en OOPD-eksaminand.

5. Desuden er alle eksaminander også underlagt citatloven og KUs ordensregler i øvrigt, herunder om videnskabelig etisk adfærd. Det betyder, at kilder til viden, der viderebringes, som er væsentlige og ikke har almenviden inden for faget, være det i form af personer, bøger, artikler, eller elektroniske ressurser, skal angives, og citater skal markeres som sådan. Bemærk, at det at kopiere en stump kode således kræver citation.

3 Opgaver

Opgaven går ud på at færdigprogrammere et spil, hvor man kan bevæge en spejder (eng. *explorer*) rundt i en labyrint (eng. *maze*). Spillet giver både mulighed for at spejderen kan vælge en retning den vil gå i, for at nå til en målposition i labyrinten, eller også kan brugeren vælge selv at bevæge spejderen i en vilkårlig retning.

Til at starte med vil spejderen af sig selv vælge den direkte vej mod et mål, og der er således risiko for, at den bliver fanget i labyrinten, da den er for dum til at gå uden om labyrintens vægge uden interaktion fra brugerens side. Undervejs i opgaven vil du blive bedt om at implementere en smartere stifinder-algoritme, som gør spejderen i stand til at finde den rigtige vej uden om forhindringer til målpositionen, hvis en sådan vej da findes.

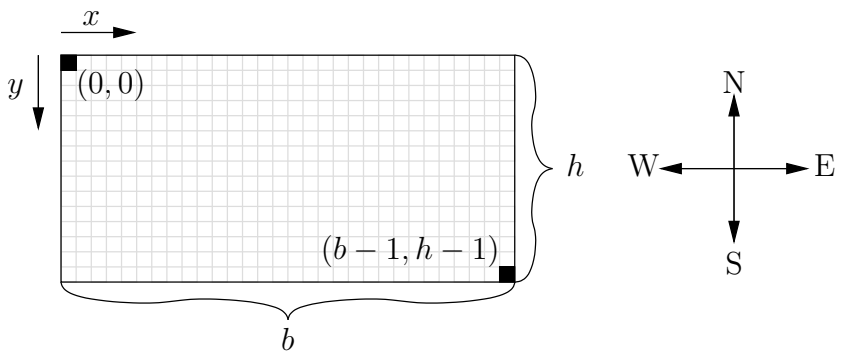
Note omkring valg af koordinatsystem

`model.Position` og `model.Direction` repræsenterer hhv. en position (dvs. et sæt koordinater) og en retning (dvs. nord, syd, øst, vest). Inden for billedbehandling er det almindeligt at y-aksen har positive stigende værdier nedad, og ikke opad som man måske er vant til. Et billedes *øverste venstre* pixel har koordinaterne (0,0), mens billedets *nederste højre* pixel har koordinaterne $(b-1, h-1)$ hvor b og h er henholdsvis bredden og højden af billedet. I den udleverede kode benyttes et tilsvarende koordinatsystem. Det betyder, at når man eksempelvis bevæger sig nordpå, vil y-koordinaten blive *mindre*. Se i øvrigt figur 1 på næste side.

Opgave 1: Strategy Pattern

Den udleverede klasse `model.Explorer` repræsenterer en spejder. En `Explorer` kan bevæge sig rundt i en labyrint, som er modelleret i klassen `model.Maze`. Man kan enten fortælle `Explorer`'en at den skal tage et skridt (dvs. bevæge sig et enkelt punkt) i en given retning (vha. `Explorer.move()`-metoden), eller man kan bede den tage et skridt i den retning, den selv mener er den mest fornuftige i forhold til en givet mål-position vha. `Explorer.moveTowards()`-metoden.

I den udleverede kode er der kun implementeret en enkelt algoritme til at finde vej. Opgave 1 går ud på at ændre koden, således at den nuværende stifinderalgoritme let kan udskiftes med andre algoritmer.



Figur 1: Sammenhængen mellem koordinater, retninger og positioner i denne opgave (og inden for billedbehandling i almindelighed).

Delopgave 1.1

1. Lav et `PathFinder`-interface. Der findes allerede en kodestump til interfacet i `model.pathfinder`, så du skal blot udvide denne. `PathFinder` skal have en enkelt metode ved navn `findBestDirection`, som returnerer en `Direction` fra en givet `Position` til en anden i en given `Maze`.

2. Dokumentér dit interface med JavaDoc kommentarer og evt. kontrakter.

Delopgave 1.2

1. Implementér en klasse `GreedyPathFinder` i pakken `model.pathfinder`, som implementerer `PathFinder`-interfacet. `GreedyPathFinder` skal kunne finde vej vha. samme stifinderalgortime, som den der findes i den udleverede `Explorer.findBestDirection`-metode.

2. Lav `Explorer` om, sådan at den i stedet for at bruge sin egen implementering af en stifinderalgoritme, bruger et objekt, der overholder `PathFinder`-interfacet til at finde vej. Det skal være muligt at oprette en `Explorer` med et hvilket som helst objekt, der overholder `PathFinder`-interfacet, sådan at `Explorer`en kan bruge forskellige stifinderalgoritmer.

Delopgave 1.3

Bevar følgende spørgsmål i dit besvarelsesdokument:

1. * Hvad kunne man ellers have gjort for at facilitere muligheden for forskellige stifindingsalgoritmer, hvis man ikke benyttede et *Strategy pattern* som her?

2. * Forklar fordele og ulemper ved *Strategy pattern* evt. ift. alternativer du måtte have nævnt i opg. 1.3.1 herover.

Opgave 2: Lees algoritme

Opgave 2 går ud på at implementere en alternativ, smartere stifinder-algoritme kendt som *Lees algoritme*. Lees algoritme bruges til at lægge ledningsspor i printplader i elektronikindustrien, men i denne opgave er vi udelukkende interesseret i dens stifinder-funktionalitet.

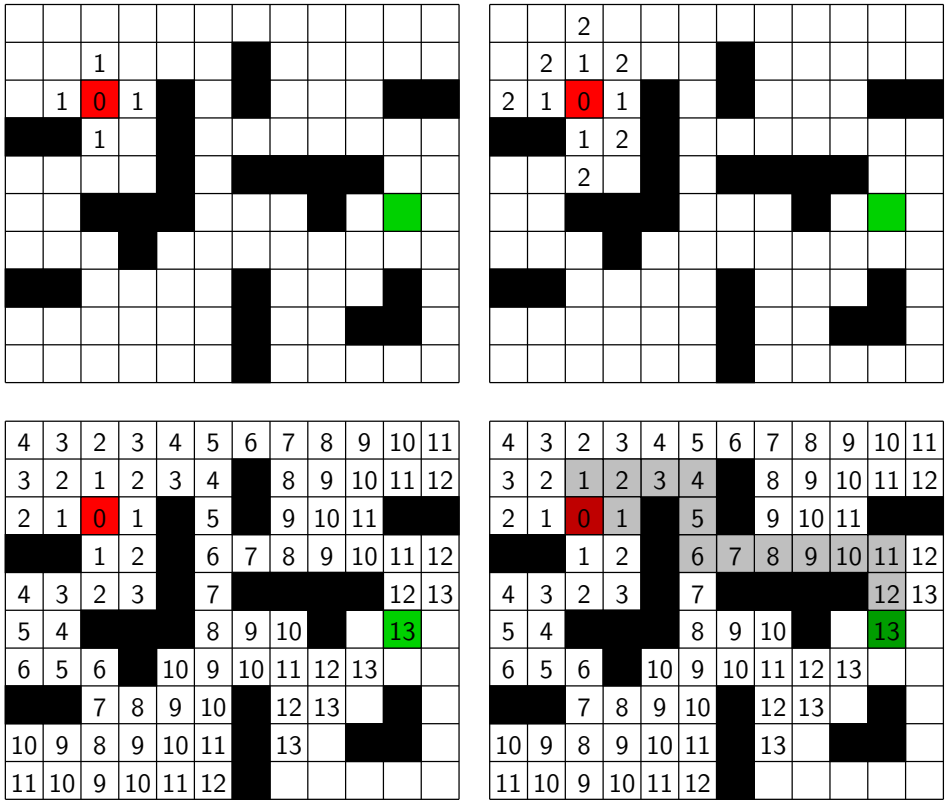
Lees algoritme er en simpel måde at finde en sti (eng.: *path*) fra et felt til et andet i et gitter af felter. Algoritmen virker på følgende måde:

1. Sæt $i = 0$.

2. Vælg et startfelt, og marker det med værdien af i.

3. Gentag følgende indtil vi enten har nået vores målfelt eller der ikke er flere tilladte nabofelter:
   (a) Sæt $i = i + 1$.
   (b) Marker alle tilladte nabofelter til felter markeret med $i - 1$ med værdien i.
4. Vælg målfeltet, og gør følgende indtil vi når startpunktet:
   (a) Vælg et nabofelt, der har en lavere værdi end det aktuelle felt.
   (b) Tilføj dette felt til stien fra startfelt til målfelt.

Algoritmen er yderligere beskrevet i figur 2 herunder, og der findes en illustrativ animation på `http://cadapplets.lafayette.edu/MazeRouter.html`.



Figur 2: Lees algoritme. Vi vil gerne finde den korteste vej fra startfeltet, markeret med rødt, til målfeltet, markeret med grønt. Vi markerer startfeltet med værdien 0. I første iteration af algoritmens skridt 3, bliver startfeltets naboer markeret med værdien 1 (figuren øverst til venstre). I anden iteration bliver naboerne til de felter, vi markerede i sidste iteration, markeret med værdien 2 (figuren øverst til højre). Dette fortsætter indtil vi når målfeltet (figuren nederst til venstre) eller indtil ingen gyldige naboer kan findes (ikke vist). Til slut finder vi fra målfeltet tilbage til startfeltet vha. skridt 4. Resultatet er en sti fra startfelt til målfelt markeret med gråt i figuren nederst til højre. Bemærk, at der er to mulige veje i eksemplet – både den, der starter over startfeltet, og den der starter til højre for det.

Delopgave 2.1

Den udleverede, ufærdige klasse `model.pathfinder.LeePathFinderGrid` skal indeholde metoder og informationer, der kan hjælpe med at implementere Lees algoritme. Fx er det meningen, at klassen

skal kunne tildele værdier til et specifikt felt, den skal kunne tildele værdier til et specifikt felts naboer, og den skal kunne returnere en `java.util.Collection` af positioner med en given værdi. Felter tildeles kun værdier hvis det er tilladt ift. den `Maze`, der findes vej inden for. Det er vigtigt at læse og forstå dokumentationen til klassen grundigt inden man går i gang med opgaven.

1. Kode klassen `LeePathFinderGrid` færdig, dvs. implementér de manglende metoder `setLeeValue()`, `getLeeValue()` og `getPositionsWithLeeValue()` samt `LeePathFinderGrid`s constructor.
2. * Lav en testplan for `setLeeValue` og `getLeeValue`-metoderne. Testplanen skal inkluderes i dit besvarelsesdokument.
3. Implementér testplanen i JUnit4. Testkoden skal lægges i pakken `test`.
4. * Konkluder på testresultaterne i dit besvarelsesdokument.

Delopgave 2.2

1. Lav en klasse `LeePathFinder` i pakken `model.pathfinder`, som implementerer `PathFinder`-interfacet fra Delopgave 1.1. `LeePathFinder` skal finde vej vha. Lees algoritme. Det kan være en god ide, at lave en uformel (dvs. ikke JUnit og ikke nødendigvis udtømmende) test af klassen, så du sikrer dig, at den opfører sig som forventet.

Opgave 3: Konfigurationsindlæsning

For at give programmet meningsfyldte data, er det nødvendigt at indlæse en konfigurationsfil. En konfigurationsfil har følgende format:

```
maze: <filnavn>
zoom: <z>
explorer: <type> <x> <y>
goal: <x> <y>
```

hvor de enkelte elementer ser således ud:

`<filnavn>`: En sti til en billedfil i `png` eller `bmp` format.[1]

`<type>`: Enten `greedy` eller `lee` afhængig af hvilken stifindingsalgoritme den resulterende `Explorer` skal bruge.

`<x>`, `<y>` og `<z>`: Heltal, som angiver henholdsvis x- og y-koordinaten af en position samt zoomfaktoren af hvad der vises på skærmen.

Der kan være et vilkårligt antal blanktegn mellem hvert element.

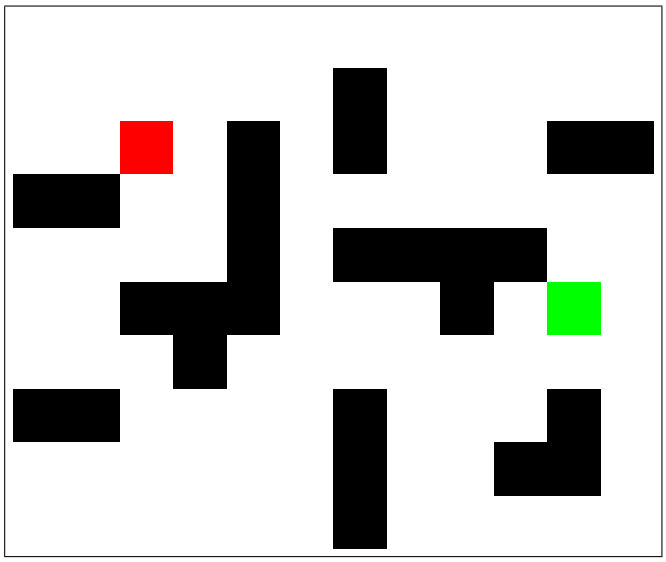
Der er flere eksempler på konfigurationsfiler og tilhørende billedfiler sammen med den udleverede kode. Eksempelvis skal indlæsning af `simple.txt` danne en configuration med data svarende til figur 3 på side 7.

Delopgave 3.1

1. Implementér `createFromConfigFile(...)` i den udleverede, ufærdige klasse `model.Configuration`. Du kan bruge den private hjælpemetode `readMazeFromImage(...)` samt den private constructor til hjælp.

[1] Det er tilrådeligt at give den fulde sti til filnavnet, da mappen, som Java som standard kigger i, kan variere fra platform til platform.

Figur 3: Konfigurationen indlæst fra `simple.txt` svarer til eksemplet brugt til at forklare Lees algoritme i figur 2 på side 5. Positionen af spejderen er rød; målpositionen er grøn. Når programmet er kodet færdigt, er det et lignende syn, der skal møde brugeren, bortset fra at der vil være ekstra knapper til at flytte den røde spejder rundt i labyrinten.



Opgave 4: Swing og Model-View-Control

Delopgave 4.1

Den udleverede klasse `view.GameDisplayPanel` er i stand til at vise et billede bestående af et gitter af farver. Størrelsen af gitteret samt hvor stor hver farvepixel skal være gives med som argumenter til `GameDisplayPanel`s constructor.

1. Opdatér `model.Explorer` sådan at den nedarver fra `java.util.Observable`-klassen. Husk at sørge for, at evt. registrerede `java.util.Observer`s får besked når `Explorer`-objektet har ændret sig.

2. Opdatér `model.Maze`-klassen på tilsvarende måde.

3. Udvid `GameDisplayPanel`, sådan at den implementerer `Observer`-interfacet. Implementér `GameDisplayPanel.update`-metoden. `update`-metoden har ansvar for at opdatere det, der vises på skærmen, og afsenderen kan enten være en `Explorer` eller en `Maze`. Du kan evt. bruge nogle af de eksisterende metoder i `GameDisplayPanel` til hjælp. Husk at kalde `paint()`-metoden når du har opdateret de relevante pixels – før det gøres, opdateres skærmbilledet ikke.

Delopgave 4.2

Den udleverede klasse `view.ControlPanel` indeholder fem knapper, hvis formål er at styre `Explorer`en i en given retning. De fem knapper kan styre `Explorer`en enten op, ned, til venstre, til højre, eller i "den rigtige retning", hvilket er bestemt af den pågældende `Explorer`s stifinderalgoritme.

1. Kod klassen `control.ExplorerController`. Klassen eksisterer allerede som kodestump og skal gøres færdig. Dens opgave består i at opdatere modellen afhængig af hvilken knap, brugeren trykker på.

Delopgave 4.3

`MainWindowFrame` skal kunne vise programmet i sin helhed, og er det brugeren ser når programmet kører.

1. Lav constructoren til `view.MainWindowFrame`-klassen, sådan så `MainWindowFrame` kan vise programmet i sin helhed.
2. Skriv `main`-metoden i `control.GameLauncher` færdig til et fungerende program – det indebærer, at koble model-, view- og control-delene sammen, og få det hele til at fungere!

Litteratur

[1] DIKUTAL.DK. *Objektorienteret Programmering og Design, semaforum @ dikutal.dk.* `http://dikutal.dk/?q=semaforum&/viewforum.php?f=102`.

[2] KØBENHAVNS UNIVERSITET. *SCI-B2-0910-Objektorienteret programmering og design.* `https://absalon.ku.dk/main.aspx?CourseID=9043`.

[3] KØBENHAVNS UNIVERSITET. *SIS - StudieInformationsSystem. Objektorienteret programmering og design, efterår 2009.* `http://sis.ku.dk/kurser/viskursus.aspx?knr=109354`.