# Labyrinter

## Algoritmer och datastrukturer

### Obligatorisk Laboration nr 6

### *Syfte*

Att ge träning i

- tillämpning av bekanta datastrukturer.
- instudering och tillämpning av en ej tidigare bekant datastruktur.

### *Mål*

Arbetet skall resultera i ett program för konstruktion och genomsökning av labyrinter. Programmet skall ha ett grafiskt användargränssnitt. Ett färdigt sådant finns så att arbetet kan inriktas på programmets interna logik.

### *Litteratur*

Ett centralt moment i laborationen är att på egen hand instudera en datastruktur som ej behandlats tidigare i kursen och tillämpa denna på ett problem. Datastrukturen i fråga kallas *disjoint set* och beskrivs i Weiss kap. 24.1-5. Kap. 14.1-3 behandlar grafer.

### *Programkod*

Givna kodavsnitt finns på kursens hemsida.

### *Genomförande och redovisning*

Uppgifterna är obligatoriska.

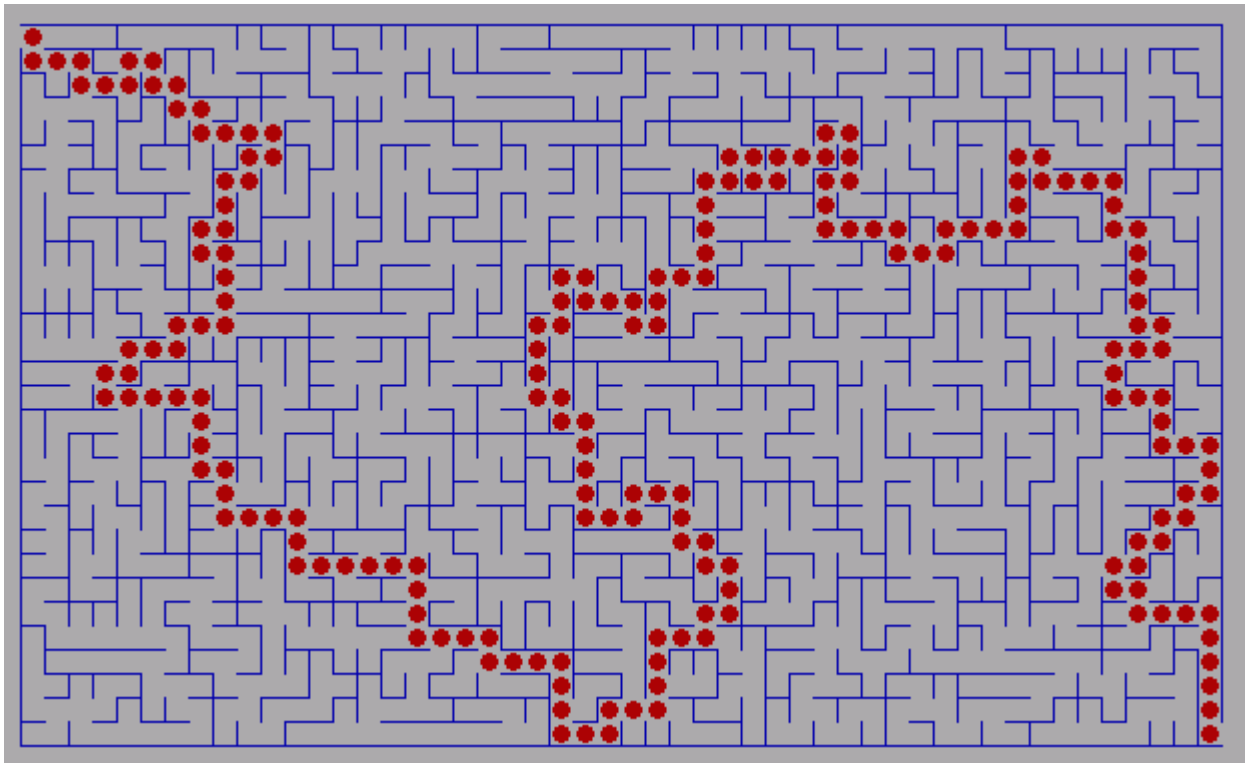
- Redovisning sker i grupper om två personer – inte färre, inte fler.
- Varje grupp skall självständigt utarbeta sin egen lösning.
  *Samarbete mellan grupperna om övergripande principer är tillåtet, men inte plagiering.*

Gå till kurshemsidan och vidare till **Laborationer->Hur man använder Fire-systemet**.

- Om du inte registrerat labbgrupp i Fire än: Gör laboration 0.0 innan du fortsätter!
- Redovisa filerna `Maze.java, ExtendedGraph.java, Gui.java` samt `BoardDispplay.java`. med dina lösningar, samt `README.txt` med allmänna kommentarer om lösningen.
- Andra halvan av **Redovisningsproceduren** beskriver hur du postar materialet i Fire.
- Senaste redovisningsdag, se Fire.

### *Inledning*

Uppgiften går ut på att konstruera och genomsöka tvådimensionella labyrinter som exemplet i bilden nedan visar. Labyrinterna skall vara rektangulära och ha slumpmässigt valda förbindelser mellan rummen (cellerna). Labyrint och genomsökning skall visas grafiskt i ett ritfönster. Konstruktionsproblemet beskrivs översiktligt i Weiss avsnitt 24.2.1. Det krävs att du är väl insatt i problemet innan programmeringsarbetet påbörjas.



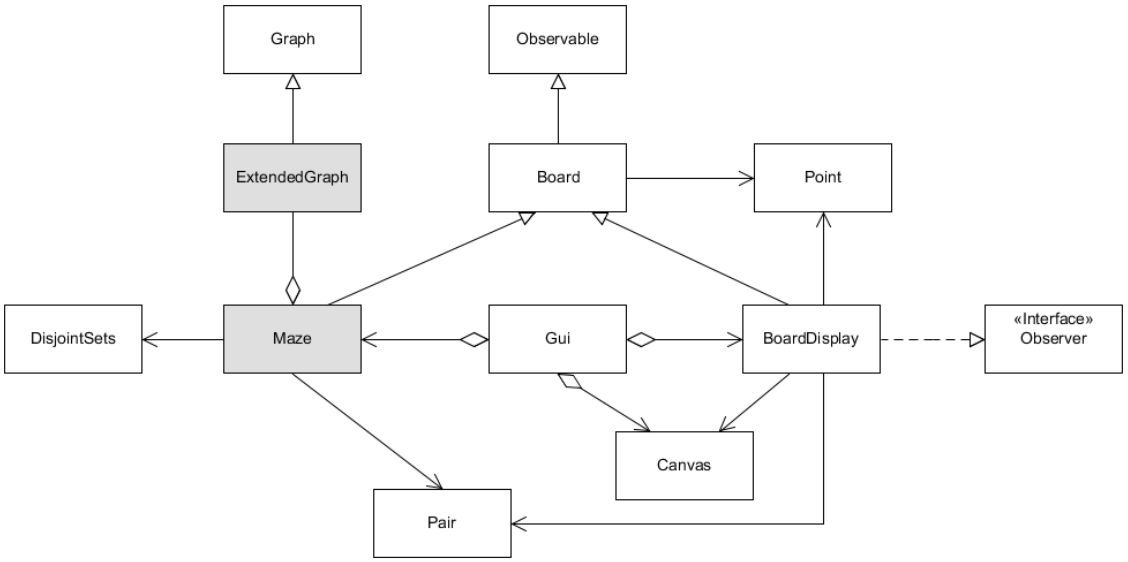
### *Krav*

En korrekt konstruerad labyrint skall uppfylla följande:

1. Det skall finns exakt en ingång längst upp till vänster och en utgång längst ner till höger som bilden ovan visar.
2. Varje cell skall kunna nås från vilken annan cell som helst.
3. Det får inte finnas några cykliska vägar.
4. Labyrinten skall presenteras grafiskt som exemplet visar.

### *Modell*

En grundmodell för programmet bör bygga på en separation av de underliggande klasserna som används vid konstruktion och genomsökning av labyrinten från den grafiska presentationen av labyrinten på skärmen. Såväl konstruktionen som den grafiska presentation bygger dock på grundförutsättningarna att en labyrint har en viss given bredd och höjd, samt en relation mellan en cells löpnummer och dess rad- och kolumnindex. För att både datastrukturdelen och grafikdelen skall kunna ha denna kunskap finns den givna klassen `Board` som definierar grundläggande dimensions- och koordinategenskaper. Modellen för programmet ges av följande UML-diagram:




För att isolera den del av programmet som innehåller själva logiken för konstruktion och genomsökning av labyrinten från det grafiska användargränssnittet skall applikationen tillämpa designmönstret Observer (se sid. 5).

I algoritmerna är det oftast bekvämast att identifiera cellerna med ett löpnummer. För den grafiska presentationen är det däremot enklare att ange koordinater med rad- och kolumnindex. Sambandet mellan dem framgår av följande exempel. Den gråmarkerade cellen har t.ex. cellID = **6**, radindex = *1* och kolumnindex = *2*.

| | *0* | *1* | *2* | *3* |
|---|---|---|---|---|
| *0* | **0** | **1** | **2** | **3** |
| *1* | **4** | **5** | **6** | **7** |
| *2* | **8** | **9** | **10** | **11** |

Metoder för konvertering mellan cellID och koordinater finns i klassen `Board`.

Alla klasser är färdiga utom `Maze` och `ExtendedGraph`. De färdiga klasserna beskrivs närmare i bilagan sist i PM.

## *Implementering*

### Programkod

Given programkod finns på kursens hemsida. Ett påbörjat BlueJ-projekt innehåller ett färdigt grafiskt användargränssnitt samt en del hjälpklasser. Metoden `main` finns i klassen `Gui`. Projektet är kompiler- och exekverbart och visar gränssnittet på skärmen.

### Arbetsgång

1. Läs in de angivna litteraturavsnitten.
2. Testa några av de givna komponenterna. Det kan underlätta lösningen av uppgiften.
3. Konstruera labyrintprogrammet. Bygg ut stegvis och testa noga i varje steg.

### Testa komponenter

Testa grafikklassen `BoardDisplay`. Skriv t.ex. en metod som slår ut ett antal slumpvis valda väggar med `knockDownWall`. Testa även `fillCell`, som ritar en röd punkt i en cell. Fyll t.ex. en diagonal av celler med punkter. Utnyttja basklassens metoder för hantering av positioner. Provkör gärna Weiss:s testprogram för klassen `DisjointSets` för att öka din förståelse för klassen.

### Konstruera labyrintprogrammet

Vi skall lösa problemet i två steg. Den första resulterar i ett program som kan rita upp en slumpmässig labyrint på skärmen, men inget mer. I det andra steget lägger vi till funktionalitet så att den kortaste vägen genom labyrinten beräknas och visas på skärmen.

*Steg 1 Konstruktion av slumpmässig labyrint*

Grundprinciperna för labyrintkonstruktion beskrivs i Weiss 24.2.1. Den beskrivna metoden räcker för att konstruera en labyrint på skärmen, så vi börjar med det. Man ritar upp ett rutnät av väggar och tar sen bort väggar under beräkningens gång. All kommunikation mellan programmets logik och det grafiska användargränssnittet skall ske med Observer-mönstret (se nästa sida). Först inför vi klassen `Maze` som skall vara en subklass till `Board`:

```
public class Maze extends Board {
    public void create()
        Skapar en labyrint på skärmen enligt metoden i kursboken.
    public void search()
        Tar reda på kortaste vägen genom labyrinten från ingång till utgång
        med lämplig  grafalgoritm, samt presenterar vägen grafiskt i ritfönstret.
}
```

Skriv `create` först, `search` utvecklas i steg 2. Fortsätt inte med nästa steg förrän labyrinten ritas upp korrekt på skärmen. Testa noga, både små och stora labyrinter, minst 400 x 400, men även specialfall som 2 x 1, 1 x 2, 2 x 2, 1 x 10, 10 x 2, o.s.v.

*Steg 2 Beräkning av väg genom labyrinten*

För sökning efter en väg genom labyrinten behövs ytterligare en datastruktur. En metod är att samtidigt med den grafiska labyrintkonstruktionen i `create` bygga en motsvarande oriktad graf. Varje cell i labyrinten motsvaras av en nod i grafen och en utslagen vägg mellan två celler av en båge i grafen. Om alla labyrintceller är nåbara från varandra, och labyrinten saknar cykler, så kommer grafen att bli ett orotat träd. Den givna grafklassen är avsedd för riktade grafer, men oriktade bågar kan lätt simuleras genom att alltid göra två parvis motriktade bågar. Utvidga grafklassen genom att definiera följande subklass:

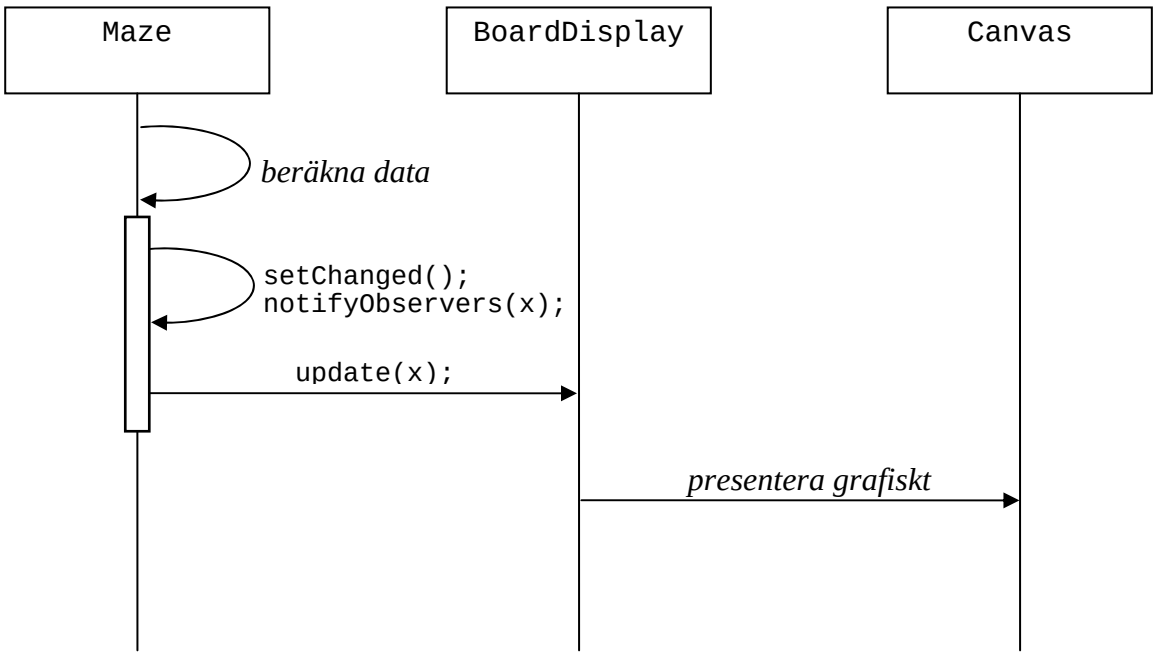
```
public class ExtendedGraph extends Graph {
    public List<Integer> getPath( int destName ) { ... }
    private List<Integer> getPath( Vertex dest ) { ... }
}
```

Metoden `getPath` skall returnera en lista innehållande den senaste vägen från start- till destinationsnod som hittades med grafsökningsalgoritmen. Metoden behövs för att senare kunna rita en väg genom labyrinten på skärmen. Den privata metoden skall vara rekursiv och fungera självständigt utan hjälp av extra instans- eller klassvariabler. *Tips:* i bas- klassen finns funktionen `printPath` som delvis kan användas som förebild. Testa genom att konstruera någon av graferna i kursboken, anropa sökalgoritm för kortaste väg mellan två noder, hämta vägen med `getPath`, och skriv ut den.

Labyrintgrafen analyseras sedan med lämplig grafsökningsalgoritm. Presentation av den funna vägen kan göras med `fillCell` som ritar röda punkter i labyrinten på skärmen (se nästa sida).

**Observer-mönstret**

När labyrinten konstrueras beräknas i varje steg vilken vägg som skall tas bort. Denna kan beskrivas med en cellposition och en riktning, t.ex. betyder cell nr 123 och riktning UP att den vågräta väggen ovanför cell nr 123 skall tas bort. Under den efterkommande sökfasen beräknas i varje steg en position på den funna vägen genom labyrinten som skall markeras med en röd punkt. För att `update` i `BoardDisplay` skall kunna veta vilket fall som inträffat kan man skicka med en parameter till `notifyObservers` som beskriver detta. Parametern skickas vidare till `update` som anropas automatiskt av `notifyObservers`. I det första fallet skall parametern ha typen `Pair<Integer,Point.Direction>` och i det andra `Integer`. Ett sekvensdiagram beskriver det hela:




### *Utvecklingsmöjligheter*

Den föreslagna uppgiften har väl sina begränsningar. Man kan tänka sig en mängd utvidgningar av programmet, t.ex. någon form av labyrintspel:

- Låt användaren söka manuellt, t.ex. med piltangenterna, i labyrinten och ge poäng baserat på antal besökta rum, tidsåtgång etc.
- Presentera labyrintens rum på skärmen först när de besöks för att begränsa användarens överblick.
- Presentera lösningen (som i basuppgiften).
- Logga poäng i fil.
- Tillåt cykliska vägar.
- Dörrar kan slå igen och nya öppnas.
- Monster kan dyka upp.
- Egna idéer?

## Färdiga klasser

Klasserna nedan är färdiga. Beträffande `List`, `DisjSets`, `Graph` mf.l. se kursboken. I projektet används en förenklad version av klassen `Graph` vilken bifogas. Inga av de givna klasserna får ändras!

### `class Point`

Används för att representera punkter i det tvådimensionella planet.

`enum Direction {UP,RIGHT,DOWN,LEFT}`
Används för att representera riktningar.

**Relationer**
Objekt av klassen förekommer på ett flertal ställen i programmet.

**Metoder**
`public Point(int r,int c)`
Konstruktor. Initierar objektet med rad- och kolumnindex.

`public Direction getDirection(Point p)`
Returnerar den lod- eller vågräta riktningen till `p`. Funktionen ger bara korrekt information om den anropas för punkter som ligger i det anropande objektets våg- eller lodlinje.

`public void move(Direction d)`
Ändrar punktens koordinater ett steg i riktning `d`.

**Instansvariabler**
`private int row`
Punktens radindex.
`private int col`
Punktens kolumnindex.

### `class Board`

Representerar grundläggande dimensionsdata om en labyrint.

**Relationer**
Basklass till `Maze` och `BoardDisplay`.

**Metoder**
`public Board(int m,int n)`
Konstruktor. Initierar objektet med labyrintens dimensioner.

`public int getcellId(Point p)`
Returnerar `p`:s cellId

`public int getRow()`
Returnerar `p`:s radindex.

`public int getCol()`

Returnerar `p`:s kolumnindex.

**`public boolean isValid(Point p)`**

Returnerar `true` om `p` är inom labyrintens gränser, `false` annars.

**Instansvariabler**

**`private int maxRow`**

Maximalt antal rader numrerade `0..maxRow – 1.`

**`private int maxCol`**

Maximalt antal kolumner numrerade `0..maxCol – 1.`

**`private int maxCell`**

Maximalt antal celler numrerade `0..maxCell – 1.`
Vairiabeln initieras i konstruktorn till `maxRow*maxCol.`

## `class BoardDisplay`

Används för att rita ut labyrinten i ritfönstret, samt för att markera en väg genom labyrinten.

**Relationer**

Publik subklass till `Board`. Implementerar gränssnittet `Observer`. Har ett observer-förhållande till `Maze`. Erhåller data från `Maze` som presenteras grafiskt i `Canvas`.
OBS! Samtliga metoder utom `update` är med avsikt deklarerade som privata. Det är alltså meningen att de skall anropas från `update` – inte från `Maze`!

**Metoder**

**`public BoardDisplay(Canvas canvas,int m,int n)`**

Konstruktor. Initierar objektet med labyrintens dimensioner.
Räknar även ut labyrintens positionering i ritfönstret.

**`private void drawGrid()`**

Ritar upp ett grundläggande rutmönster med alla väggar intakta i labyrinten.

**`private void knockDownWall(int cellId,Direction dir)`**

Tar bort en cellvägg i ritfönstret.

**`private void fillCell(int cellId)`**

Ritar en besöksmarkering i angiven cell i ritfönstret. Används vid Presentation av en väg genom labyrinten.

**`public void update(…)`**

Tar emot data från Maze om vägg som skall rivas, eller position i labyrintsökning och presenterar dem grafiskt på lämpligt sätt. Metodstump finns.

**Instansvariabler**

Klassen har en mängd interna variabler som behövs för grafiken.

### `class Pair<A,B>`

Används t.ex. för att representera cell/granncell-par vid genereringen av labyrinten.

### `class Canvas`

**Relationer**

Används av `BoardDisplay` för presentation av labyrinten i GUI:t.

Klassen är en modifierad version av en liknande klass i Barnes och Köllings bok Objects first with Java.

### `class Gui`

**Relationer**

Skapar en `Canvas` för labyrintgrafiken. Skapar objekt av `Maze` och `BoardDisplay`. Gör `BoardDisplay` till observatör av `Maze`. Skapar det grafiska användargränssnittet. Observera att inmatningen av heltal i textfälten för bredd och höjd måste avslutas med Return i respektive fält. Viss indatakontroll av inmatade data sker.

*Lycka till!*