Semesteroppgave: dynamiske brett

DATS1600 oppgavesett 6.

I dette oppgavesettet skal dere bruke dynamiske datastrukturer fra Javas Collections biblioteket. Ved bruk av slike datastrukturer, skal dere omgjøre GoL spillebrettet til å være *dynamisk*.

Generisk programmering og Java Collections er presentert i forelesning i uke 11, 12, og 13 og en oppfrisker tutorialtime holdes i uke 17.

Dynamiske brett

Deres nåværende brett er definert via en statisk 2D tabell (statisk som i det norske ordet 'statisk', ikke som i Javas **static** nøkkelord). For eksempel, hvis et mønster er større enn lengden (i vidde eller høyde) til brettet, kan ikke hele mønsteret lastes inn på brettet. For å støtte en slik funksjonalitet må vi kunne tilby *dynamiske* brett, som kan dynamisk endre sin størrelse etter at brettet har blitt opprettet.

For å tilby en slik funksjonalitet skal dere implementere brettet via en dynamisk datastruktur. Til sammenligning, er Java tabeller en form for en statisk data struktur: dens størrelse kan ikke endres etter at tabellen har blitt opprettet.

Merk at dere skal ikke *erstatte* deres tidligere implementasjon av spillebrettet. Dere skal istedenfor implementere en *ny* representasjon av spillebrettet. Etter at dere har utført denne oppgaven, vil dere dermed tilby to implementasjoner av spillebrettet. Diskuter hvordan disse to implementasjonene skal håndteres i deres kodestruktur.

Hvis deklarasjonen av deres 2D brett i deres nåværende implementasjon er:

```
private byte[][] board;
```

skal dere nå opprette en ny klasse, der dere deklarerer brettet som følgende:

```
private List<List<Byte>> board;
```

der List er interfacet java.util.List. Merk at dette List interfacet er et eksempel på designmønsteret Strategy, der ulike implementasjoner av dynamiske lister tilbys av Java. Jeg anbefaler at dere bruker implementasjonen som implementerer listen som en Java tabell (array). Den konkrete klassen som tilbyr denne implementasjonen heter ArrayList. For interesserte studenter, kan Internett brukes til å finne ut hva LinkedList er og hvorfor ArrayList tilbyr bedre ytelse enn LinkedList for vår applikasjon (der vi primært aksesserer elementer, ikke sorterer etc.).

List er et eksempel på *generisk* programmering. Generelt, er generisk programmering et alternativ til polymorfisme. Diskuter fordelene og ulempene med generisk programmering og hvorfor List er implementert via generisk programmering. Hvordan kan List implementeres ved å bruke polymorfisme istedenfor generisk programmering?

Diskuter om det er naturlig å konvertere en eller flere klasser dere har implementert til generiske klasser. I tillegg, diskuter den generelle bruken av generisk programmering: når er det naturlig å velge denne fremgangsmåten fremfor bruk av polymorfisme?

Støtte for større brett

Anta at vi har følgende metode for å sette tilstanden til en celle:

```
public void setCellState(int x, int y) { … }
```

For den statiske implementasjonen av brettet, burde metoden kaste en exception hvis cellen (x,y) er definert utenfor brettet, siden brettet ikke kan endres etter opprettelse. IndexOutOfBoundsException eller IllegalArgumentException er begge OK i dette tilfellet.

Hovedforskjellen mellom de dynamiske og statiske brettene er at oppførselen til det dynamiske brettet er annerledes for metoden setCellState. Hvis cellen (x,y) er definert utenfor brettet, skal brettet automatisk utvides slik at brettet inneholder cellen (x,y).

For en 1D liste av typen List er slik automatisk utvidelse trivielt fordi listen kan iterativt utvides, et element om gangen, ved bruk av add metoden, helt til listen er stor nok. Les om add i Javas dokumentasjon:

https://docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-

For 2D lister (dvs. lister inne i lister), må man påse at man utvider alle lister slik at 2D listen er rektangulær (har samme antall kolonner for hver rad). Diskuter og eventuelt kladd hvordan dette skal gjøres slik dere håndterer deres 2D brett korrekt.

Opprett en JUnit test som både setter verdier innenfor og utenfor det originale brettet. Eksempel på testkode, der vi antar at størrelsen på brettet er angitt i konstruktøren:

```
GameBoardDynamic board = new GameBoardDynamic(10,10);
board.setCellState(5, 5, true);
org.junit.Assert.assertEquals(board.getCellState(5, 5), true);
board.setCellState(50, 50, true);
org.junit.Assert.assertEquals(board.getCellState(50, 50), true);
org.junit.Assert.assertEquals(board.getCellState(49, 49), false);
org.junit.Assert.assertEquals(board.getCellState(51, 51), false);
```

Etter at setCellState metoden er korrekt implementert, kan dere oppdatere kode som er relatert til denne metoden i spillet. For eksempel, vil det være naturlig å se på koden som laster inn et mønster fra en Reader stream.

Etter at dere har implementert deres nye brett-klasse, annoter den gamle brett-klassen (med 2D arrayen) som **@deprecated**. Java kompilatoren vil nå produsere en warning hvis denne klassen brukes i koden. Hvis dette er tilfellet, burde koden oppdateres slik at den nye brett-klassen brukes istedenfor.

Noen mønstre, som Gosper Glider Gun, utvider området mønsteret er tegnet på uendelig. Dere må selv vurdere om dere ønsker å utvide brettet ettersom et mønster utvider seg. Hvis dere velger å utvide brettet dynamisk, så kan det være naturlig å sette en maksstørrelse på brettet slik at programmet ikke ender opp med å bruke for mye minne.

Merk: over har vi gjort noen antagelser i kodestrukturen, som at vi har setCellState og getCellState metoder og at vi har egne klasser for brett. Hvis dette ikke er tilfellet i deres kodestruktur, så må dere justere utførelsen av oppgaven etter deres struktur.

Valgfri ekstraoppgave

Den originale implementesjonen representerer brettet via en 2D Java tabell. Denne representasjonen er faktisk tilnærmet identisk med List sin interne representasjon, antatt at dere bruker ArrayList. Det er derfor ikke problematisk å endre på den originale klassen slik at den har dynamisk oppførsel. Gjør endringer i setCellState metoden til den originale brett-klassen, slik at den har identisk oppførsel som set-metoden i den nye klassen. Utførelse av denne utvidelsesoppgaven vil gi innblikk i hvordan add-metoden i ArrayList er implementert.