Dat109 Slange og stige spill

30.01.2023

Emne ansvarlig:

Per Christian Engdal

Prosjekt medlemmer:

Ørjan Knudsen, Markus Nedrevold, Martin Lenes og Marius Larsen

Innledning

Dette er første obligatoriske oppgave i dat109. I denne oppgaven skal vi bruke UML-modeller, GRASP-prinsipper og Java for å lage “Slange og Stigespillet” som Java-app.

Hoveddel

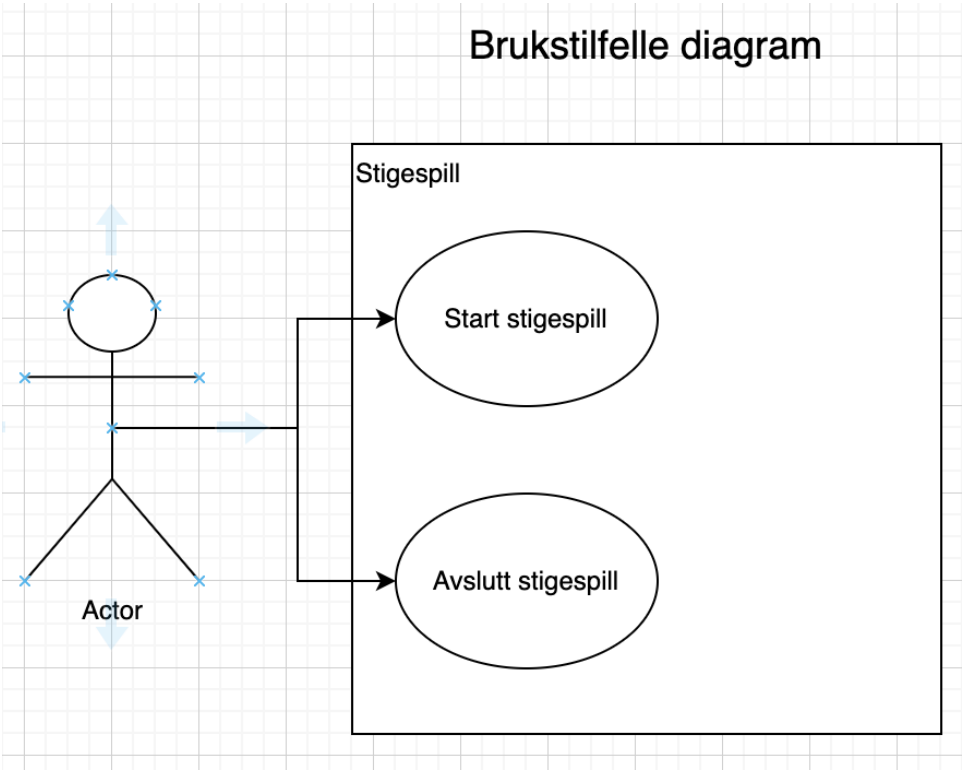
Tanken i dette prosjektet var å bruke objekt orientert design som UML og utviklings prinsipper som GRASP for å skape et veldefinert objekt orientert applikasjon for brett spillet slanger og stiger.

De viktigste prinsippene å følge var “low coupling” og “high cohesion”, som betyr at applikasjonen skal ha så lite relasjoner mellom klassene som mulig, eller at relasjonene er veldefinert og logiske.

Brukstilfellemodell

Brukstilfellediagram:

Dette diagrammet viser hvilke funksjoner systemet skal ha, og hvilke aktører som trenger de ulike funksjonene.

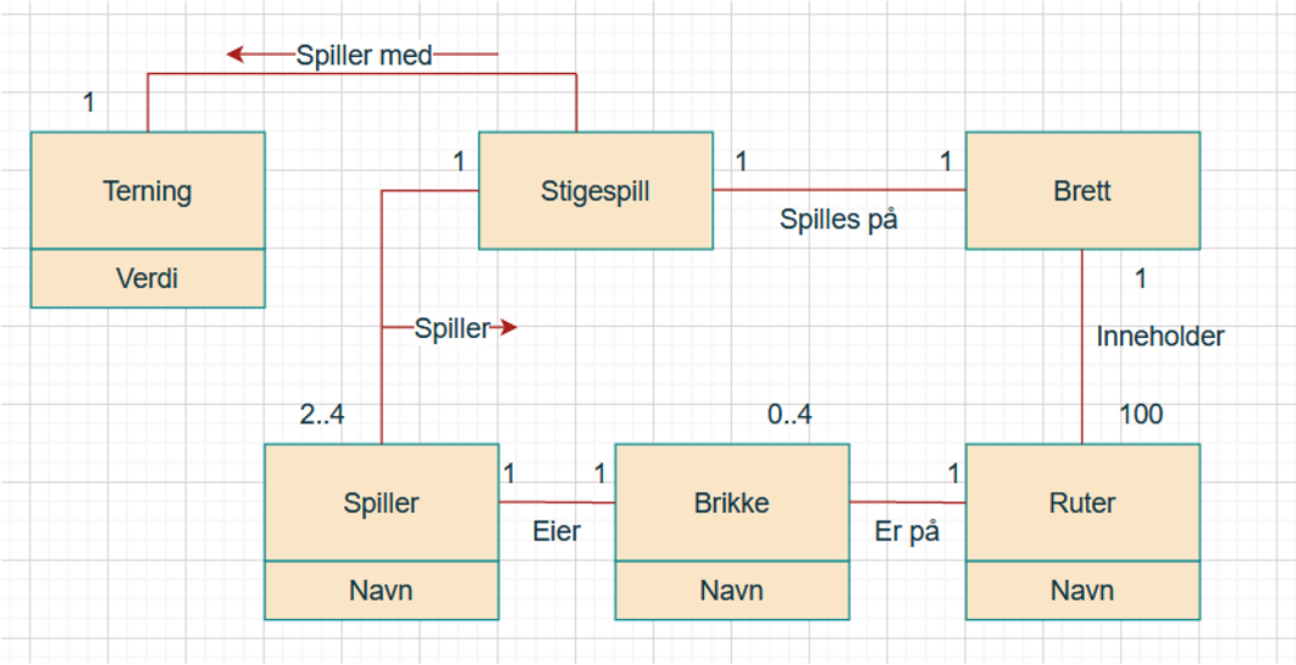


Brukstilfellebeskrivelser:

Brukstilfelle beskrivelse	
navn	spill stigespill
scope	stigespill
nivå	brukerfunksjon
primær aktør	obesravtør
interessenter	
start kriterier	
stopp kriterier	
hovedflyt	1. observatør velger nytt spill og antall spillere 2. system validere angitt data og tilbyr mulighet til å starte spillet 3. obsevatør starter spillet 4. systemet starter spillet og følger reglene 5. gjentar punkt 4 til spillet er over eller observatør avslutter spillet

Domenemodell

En domenemodell er vanligvis implementert som en objektmodell i et lag som bruker et lavere lag med utholdenhet og publiserer et API for å gi et høyere lag med tilgang til modellens data og oppførsel. I UML bruker vi et klassediagram (vises senere i teksten) for å representere domenemodellen. Under vises domenemodellen vi har brukt i vår oppgave. Hvis vi begynner på ruten der det står Stigespill. Går vi til venstre kommer vi til Terning og under vises verdien. Går vi ned til spiller viser det at i et spill kan det være 2-4 spillere (navnet til spillerne kommer under). Det er 0-4 brikker som skal stå på én rute. En spiller skal ha én brikke. Et brett har 100 ruter.



Sekvensdiagram

Sekvensdiagrammet viser hvordan objekter samhandler, dvs. i hvilken sekvens objekter utfører metodekall på hverandre. Målet er å illustrere oppførsel i et tenkt tilfelle, ved kjøring av systemet.

Sekvensdiagrammet kan leses som følger:

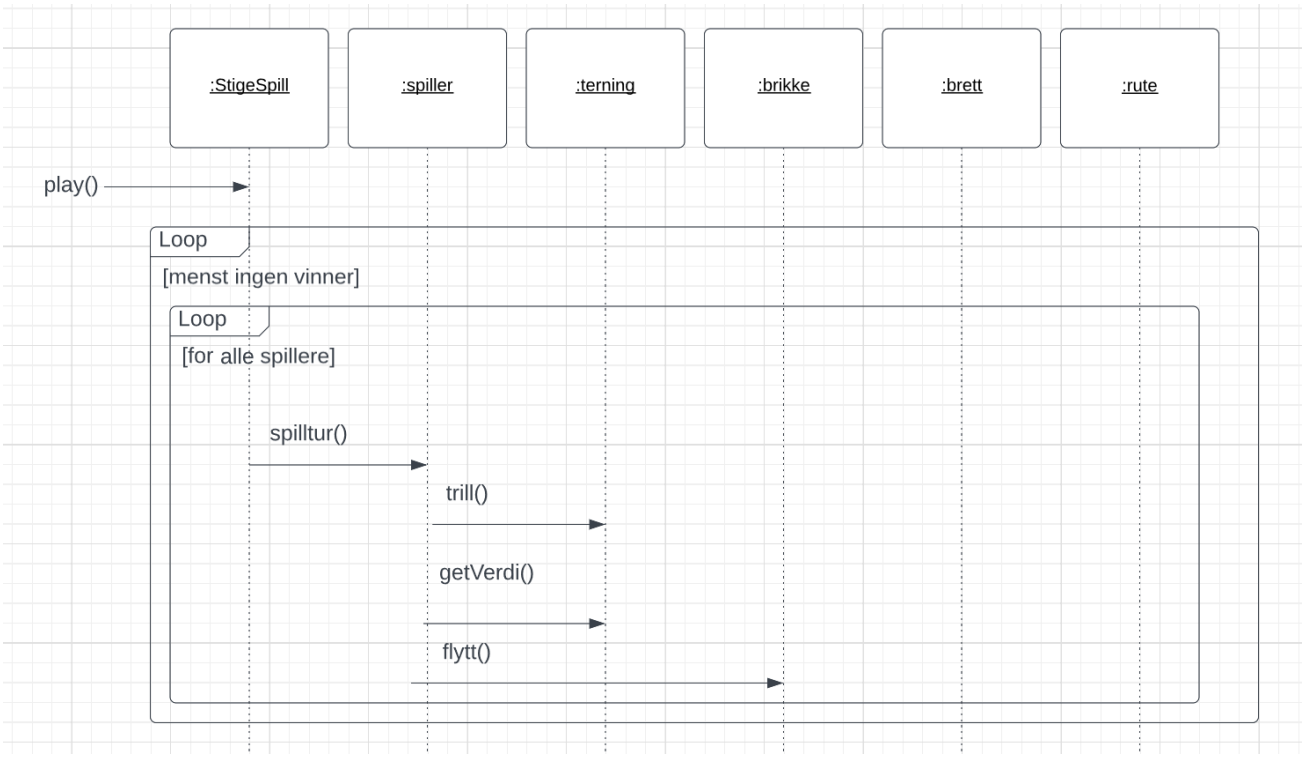
-Metoden play() starter spillet

-spilltur() avgjør hvem sin tur det er

-Spiller triller terning, får en verdi og flytter deretter

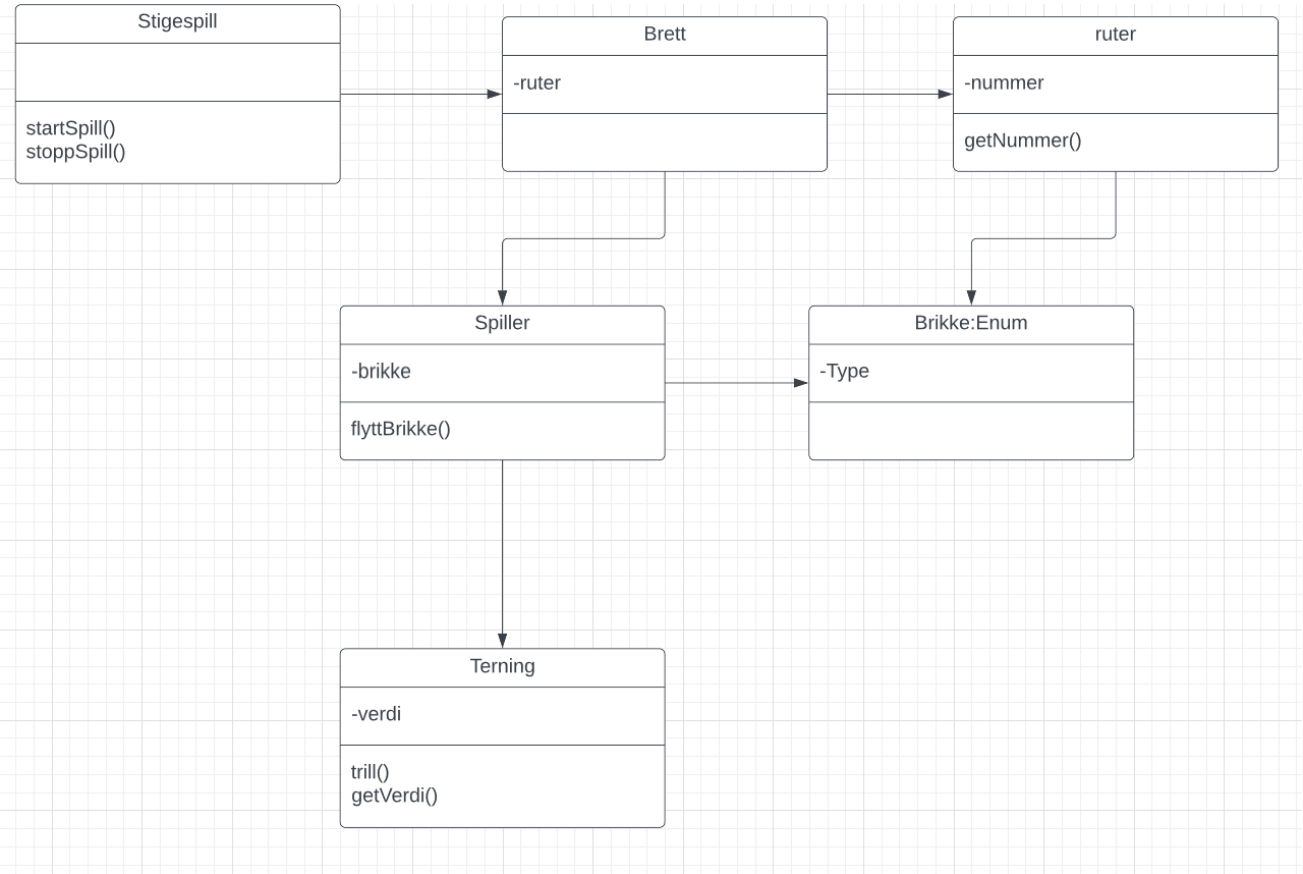
-Stigespillet fortsetter til en vinner

-Viser hvilke klasser som har ansvar for metodene



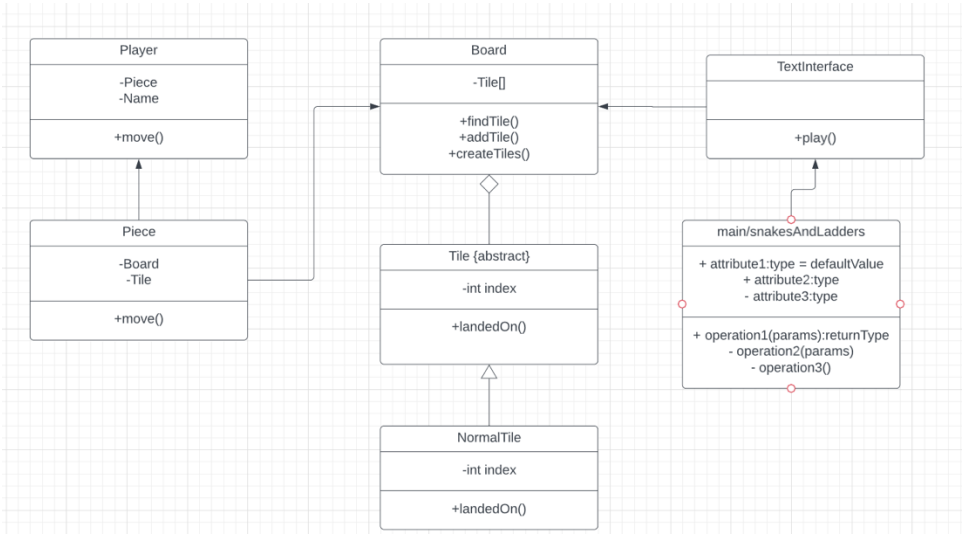
Klassediagram

Klassediagrammet viser hvordan de forskjellige klassene er avhengig av hverandre, og hvilke attributter og metoder som hører til hver klasse.



Avslutning

Klasse diagram av den endelige løsningen:



Grensesnittet TextInterface har all av logikk for metoden play(), den oppretter et brett som har en metode createTiles.

```
public void createTiles() {
    for (int i = 0; i < tiles.length; i++) {
        addTile(new NormalTile(i), i);
    }
}
```

CreateTiles fyller brettet opp med 100 ruter, i denne implementasjonen er ruter en abstrakt klasse som har en privat verdi (index) og en public metode landedOn.

```
public abstract class Tile {
    private int index;
    public void landedOn() {}

    public int getIndex() { return index; }
```

```
@Override
public void landedOn() { System.out.println("landed on a normal tile"); }
```

Metoden landedOn beskriver hva som skjer, siden denne implementasjonen har kun normale ruter, så skriver programmet bare ut en setning at spilleren landet på en vanlig rute. Neste oppgave hadde vert å implementere slanger og stiger som ruter. I neste steg av play() så oppretter den spillere og brikker basert på bruker input, denne fabrikkeringen skjer i en for løkke i metoden, en bedre implementasjon hadde brukt fabrikkerings klasser, men det oppstod problemer når vi prøvde det som en løsning.

```java
System.out.println("how many players are playing?: ");
int numberOfPlayers = scanner.nextInt();

//creates a list of players using the inputted number of players
Player[] players = new Player[numberOfPlayers];
/**
 * for loop that asks for the name of each player and creates a player object
 * with the name and a piece object with the board and the first tile.
 * tried using a factory class for this at first, but encountered problems with the
 * implementation.
 */
for (int i = 0; i < numberOfPlayers; i++) {
    System.out.println("Enter name of player " + (i + 1) + ": ");
    String name = scanner.next();
    players[i] = new Player(name, new Piece(board.getTile( index: 0), board));
}
```

Etter brukeren har valgt antall spillere og navn til spillerne, så kjører programmet til en spiller lander på rute nummer 100. Metoden move() har logikk at dersom en spiller som er nær rute 100 må kaste presist tallet for å lande på rute 100, Td en spiller som står på rute 98 og kaster 3 på terningen blir stående til han eller hun kaster 2 eller 1.

```java
public void move(int sum) {
    if (tile.getIndex() + sum > 99) {
        return;
    } else {
        Tile newTile = board.findTile(tile, sum);
        tile = newTile;
        setTile(newTile);
    }
}
```

Når en spiller lander på rute 100 så blir vinneren erklært og spillet avsluttes.

Spillet simuleres i terminalen, senere iterasjoner ville ha brukt et GUI eller omdannet applikasjonen til en web-app og brukt html elementer for å spille spillet.

Applikasjonen fungerer som en basis for senere iterasjoner, vi skulle ha likt å ha fått gjennomført mer, men vi er også fornøgd med hvor langt vi kom på tiden som ble gitt.

Bilder når programmet kjører:

```
how many players are playing?:
2
Enter name of player 1:
per
Enter name of player 2:
pål
```

```
per is on tile 97 and landed on a normal tile
pål is on tile 98 and landed on a normal tile
per is on tile 97 and landed on a normal tile
pål is on tile 98 and landed on a normal tile
per is on tile 97 and landed on a normal tile
pål won!

Process finished with exit code 0
```