Beskrivelse av systemet

- **1. Hvilke interface finnes? Hva er sammenhengen mellom dem og hva brukes de til?**

Det er tre forskjellige type interface filer i dette programmet:

- inf101.v17.boulderdash.bdobjects.IBDObject
- inf101.v17.boulderdash.bdobjects.IBDMovingObject (extends IBDObject)
- inf101.v17.boulderdash.bdobjects.IBDKillable(Extends IBDMovingObject)

IBDObject filen er interface til alle objekter som er på kartet. Objekter på kartet kan være alt fra: sand, bug, spiller og steiner. Interfacet inneholder metoder som *getColor*, som henter fargen til objektet. *GetMap* snakker med *BDMap* filen, som finner ut informasjon om kartet som: høyden, spiller-posisjon, posisjon til objekter i kartet og generelt annet informasjon om kartet og deres objekter. Andre metoder henter generell posisjoner til objektene. Den største metoden i denne klassen er *step()* metoden. Som oppdaterer «status» på objektet et steg.

IBDMovingObject er interfacet for alle objekter som flytter på seg på kartet. Den er «under» -klassen til *IBDObject*, siden den arver alt fra *IBDObject*. I dette tilfellet er det *BDBug* eller *BDPlayer*. Denne klassen er ikke like «aktiv» som *IBDObjekt* klassen, siden denne kjører kun når objekter skal bevege på seg. Derfor står det prepare i navnene og en metode som heter *getNextPosition*, siden den skal gjøre noe senere.

IBDKillable er interfacet for objekter som kan drepes. I dette tilfellet er det *BDBug* eller *BDPlayer*. Siden objektet kan også bevege seg, så er Interfacet er en underklasse av *IBDMovingObject*, så den kan hente alle objekter og funksjoner fra den klassen.

- **2. Hva slags rolle spiller arv i designet til programmet?**

Arv har en av de største rollet i designet til programmet. Dette er for å skape fleksibilitet i alle objekter. Muligheten til å velge hva som skal påvirkes uten at det påvirker andre deler. Hvis dette hadde vært færre antall filer og mindre arv-designet så hadde man ikke kunne endret noe uten å påvirket andre deler programmet. I tillegg så unngår man å repetere koder flere ganger, siden alle objektene trenger koden til f.eks. posisjon og farge. *BDPlayer* og *BDBug* trenger samme koden når det gjelder bevegelse og generelt oppførsel.

- **3. Det er flere abstrakte klasser i systemet. Hva slags funksjon har de? Hvorfor er de abstrakte?**

Abstrakte klasser i programmet:

- inf101.v17.boulderdash.bdobjects.AbstractBDObject (implements IBDObject)
- inf101.v17.boulderdash.bdobjects.AbstractBDMovingObject
- inf101.v17.boulderdash.bdobjects.AbstractBDKillingObject
- inf101.v17.boulderdash.bdobjects.AbstractBDFallingObject

De abstrakte klassene i systemet har logikken bak alle funksjoner. For eksempel i AbstractBDMovingObject så viser det hvordan steinene skal falle. Med visse mellomrom før de faller helt ned, hvor langt ned osv. Dette er en klasse for kun logikken bak objekter som skal falle. Den kan ikke være en klasse i seg selv uten å implementere en ikke-abstrakt klasse. I *AbstractBDObject* så henter man alle funksjoner osv. fra *IBDObject*. Dette er for å kunne extende *AbstractBDObject* i de andre abstrakt klassene, for å ha de samme funksjonene gjennom alle abstrakt klassene. Man bruker abstrakte klasser når man vil dele kode over flere relaterte klasser eller hvis man vil bruke ikke statiske og ikke finale felt.

- **4. Hvor er hoveddelen av logikken til spillet er implementert? Få oversikt over metodene, hvor de er implementert, hvordan de kalles.**

Hoveddelen av logikken til spillet er implementert i: boulderdash.gui.BoulderDashGUI. Metoder som brukes i denne klassen er:

- BuilderDashGUI();
- Run();
- Start();
- Step();
- Handle();

I denne klassen så er det lagt til metoder så man kan framstille den grafiske delen av spillet. Man gjør dette ved hjelp av å bruke «javafx», som er allerede eksisterende metoder. Men det er også lagt til kode for å tilpasse til spillets behov.

Grunnen for dette er hoveddelen av logikken, er at main klassen starter spillet ved å kalle på metoden Run(); fra BoulderDasGUI. Som kjører kartet som er lagt inn i MapReader i main.

Start metoden starter opp alle deler av javafx og gir spillet et spillerom å jobbe på ved å legge til bakgrunnsfarge og andre visuelle bakgrunns innstillinger (font, farge, generell tekst).

Step metoden sjekker om spilleren er i livet eller ikke. Hvis det er det så skal det vise antall diamanter som tekst. Hvis spilleren er død så skal det stå: «player is dead» i stedet for.

Handle metoden «hører» etter tastaturkommandoer. Q eller Escape avslutter programmet, f er fullscreen og ellers er det bare å høre etter generelle tastetrykk.

- **5. Hva slags rolle spiller abstraksjon i dette programmet?**

Abstraksjon har en stor rolle i dette programmet. Siden hvert eneste objekt har forskjellige funksjonaliteter som trenger en logikk bak seg. Både klasser som *BDRock* og *BDDiamon* trenger abstraktklassen som gjør at de faller. Samme gjelder *BDBug* og *BDPlayer* som begge trenger klassen *IBDKillable*. Abstrakte klasser

- **6. Hvordan kunne man lagt til en ny type felt?**

Jeg tolket denne oppgaven som type felt = objekt

For å lage en ny type objekt, så må man først lage en klasse til dette objektet. Klassen må extende den type objekt det skal være (fallende, osv.). Så legger man til metoder som man vil at den skal ha bra BDObject (getColor osv.). Deretter må man legge det til det nye objektet i makeObject metoden under BDMap. Man gir den en char verdi. For å bruke objektet så kan man legge det til i et kart.

- **7. Hvordan er det implementert at en diamant faller?**

For å implementere at en diamant faller så må klassen diamant extend *AbstractBDFallingObject* som klassen *BDRock* gjør. *AbstractBDFallingObject* sjekker om objektet kan falle, så faller objektet neste gang *step()* er aktivert.

3.1 Spørsmål

- **Hvorfor trenger vi `getPosition`-metoden? Kunne vi like gjerne ha lagret posisjonen i hvert enkelt objekt?**

Vi bruker getPosition metoden til å finne koordinater til et gitt objekt. Dette er fleksibelt i forhold til at man muligens lager nye objekter og da kan man bruke den allerede eksisterende koden. Dette sparer både plass og tid.

- **Hvis vi bruker et hashmap til å forenkle jobben med å finne posisjoner, har vi laget en sammenheng mellom to av feltvariablene; grid-et og hashmap-et. Hva er sammenhengen? (Her har vi en *datainvariant*, en begrensing på datarepresentasjonen i objektet.)**

Sammenhengen mellom grid-et og hashmap-et er at griden holder på IBDObject og hashmap-et kobler et IBDObject til et posisjonsobjekt. Objektet fungerer som en søkenøkkel. Man kan sette x og y koordinatene i grid til å alltid tilsvare koordinatene som man finner i posisjonsobjektet til et IBDObject.

- **Ville vi fått en tilsvarende sammenheng mellom grid-et i BDMap og BD-objektene våre om vi lagret posisjonen i hvert enkelt objekt? Kan det være problematisk?**

Problemet med dette er at vi bruker Posisjonsobjektet som en slags søkenøkkel. Da har vi ingen måte å koble Grid-et til hashmap-et til hverandre.