2.2 Systembeskrivelse

Det er tre interfacer i *boulderdash.bdobjects; IBDKillable.java, IBDMovingObject.java* og *IBDObject.java.* Alle ruter i kartet består av et objekt. Et objekt kan være tomt, sand, monster, spiller, stein eller diamant. Alle disse objektene lages utfra interfacet *IBDObject*. Et inferface definerer alle metoder som et objekt som er instanse interfacet må ha. Disse tre interfacene er avhengige av hverandre, dvs *IBDOject* er det grunnleggende interfacet for alle objekter, og de to andre extender dette. Hvis det er et bevegelig objekt (slik som spiller, monster og stein) har det ekstra egenskaper og utvider et vanlig objekt *IBDMovingObject extends IBDObject*. Noen bevegelige objekt (monster og spiller) kan også drepes og trenger utvidede egenskaper, *IBDKillable extends IBDMovingObject*.
Vi har også abstrakte klasser. Her har vi en grunnklasse, *AbstactBDObject* som implementerer *IBDObject*. Arv spiller en sentral rolle. Ordet *extends* ovenfor betyr at klassen arver fra en annen klasse. Klassen vil da arve alle offentlige metodene fra den klassen den *extender* fra. Dette er viktig for om flere klasser inneholder samme metoder (slik de forskjellige objekt typene (sand, stein etc.) gjør) og vi skal endre en metode, trenger vi bare å endre et sted, nemlig den opprinnelige klassen. Den opprinnelige klassen som blir *extendet* fra kalles superklassen.
Arv blir brukt i flere andre klasser slik som *BDPlayer extends AbstractBDMovingObject implements IBDKillable*. Her arver *BDPlayer* fra den abstrakte klassen *AbstractBDMovingObject* som igjen implementerer interfacet *IBDKillable.* Man kan ikke lage (direkte) instanser av abstrakte klasser, de kan bare blir arvet (extended). Vi kan lage instanser av klassen som arver fra den abstrakte klassen. Et MovingObjekt kan være flere ting, player, rock etc. Det er da naturlig at vi ikke vil kunne lage en instanse av et MovingObjekt, fordi vi vet ikke helt hva det er (det kan være flere ting). Derfor setter vi klassen til å være abstrakt, og lager en klasse for et konkret objekt (f.eks. *BDPlayer*) som kan arve fra denne abstrakte klassen. Abstrakte klasser har abstrakte metoder. De blir ikke definert i den abstrakte klassen, men i klassen som arver fra den.
Abstrakte klasser og interface er er ganske likt. En viktig forskjell mellom Interface og Abstrakte klasser er at i førstnevnte kan vi ha konkrete metoder (med innhold) i tillegg til abstrakte metoder. Interfacet inneholder kun abstrakte metoder (ingen implementering) og er dermed 100% abstrakt. En fordel med å bruke abstrake klasser er at vi kan ha konkrete metoder også (som er ferdig implementerte). Disse metodene kan, men trenger ikke å brukes av klasser som arver den abstrakte klassen. Vi bruker abstrakte klasser når vi vet (eller delvis vet) implementeringen videre av underklasser som arver.
De abstrakte klassene setter grunnmuren for logikken. F.eks. i *AbstractBDFallingObject* er logikken for fallende objekter som er helt essensiell for spillet. Her finner vi hovedlogikken. I de abstrakte klassene blir logikken som går igjen overalt beskrevet. De mer spesifikk logikk finner vi i de spesifikke klassene for de forskjellige objektene. Helt essensiell logikk finner vi i *BDPlayer.java* som er det viktigste objektet (utenom *BDMap*) siden det er spilleren som lar oss spille. Her tar vi input fra tastaturet, og vi har en step metode som vil bestemme hva som skjer videre avhengig om hvor spilleren går.

Herfra kan vi kalle på f.eks. en push metode som er implementert i *BDRock* og lar oss flytte på steiner. I *BDBug* finner vi logikk angående bevegelsesmønsteret til monsterene. Noe viktig logikk blir plassert i *BDMap.java* hvor vi f.eks. bestemmer om et objekt kan gå en bestemt retning (*canGo* metoden). Vi har også en step metode som gjør en iterasjon for alle objekter i gridet. Alle objekter har sin egen step funksjon (i falling objekts er denne i den abstrakte klassen). Vi gjør iterasjonene i spillet via *BDMap*. For å bevege objekter bruker vi et *Position* objekt knyttet til objektet. Denne har metoden *moveDirection(Direction dir)*, som endrer posisjonen i en retning dir. Flyttingen til hver objekt skjer med metoden *preparemove* som kommer fra *AbstractBDMovingObject*. Alle disse flyttingene blir realisert ved neste step.
Abstraksjon spiller en sentral rolle. Alle spesifikke objekter slik som BDPlayer, BDRock etc arver fra disse abrakte klassene. Det er fordel å ha slik arv, spesielt når vi har et større spill. F.eks. vi kunne hatt mange forkskjellige typer monster med litt ulike eganskaper. Da blir metoder og variabler som blir brukt om igjen samlet i enkelte stedet og koden blir veldig oversiktlig.
Man kan legge til et nytt felt ved å lage et nytt type objekt. F.eks. *BDSuperBug extends AbstractBDMovingObject implements IBDKillable* hvis vi f.eks. skal ha et nytt monster som kan bevege seg og kunne drepes. Vi kunne også i *BDSuperBug* gitt monstered nye egenskaper ved å lage nye metoder. F.eks. kunne vi gitt nye tilfeldige bevegelsesmønstre eller egenskap til formering.
At en diamant faller blir implementert i *AbstractBDFallingObject*. Siden diamant er av denne typen, arver den *public void fall()* metoden.

3.1 getPosition

Vi trenger getPosition metoden fordi vi må sjekke posisjonene til et objekt. Hvert objekt kan er knyttet opp til *BDMap* slik at vi kan finne posisjonen til objektet ved å buke *owner.getPosition(this)* hvor *owner* referer til instansen av *BDMap* og *this* er objektet (f.eks. en instanse av *BDBug*) vi vi kaller metoden fra.
Om vi hadde lagret alle posisjonene i hvert enkelt objekt hadde koden blitt mer rotete og komplisert, men det ville latt seg gjøre. Vi ville fortsatt trengt en metode for å finne ut hvilke objekter som er i en gitt posisjon, som nå er get() funksjonen. Denne opererer med å hente ut objektet fra gridet, gitt posisjonen. Det ville blitt litt problematisk om vi kun hadde disse posisjonene lagret i hver objekt.
Når vi representerer gridet som hashmap har vi laget en relasjon der hvert grid punkt sitt objekt og posisjon er en input i et hashmap. Hvert element i hashmapet har en nøkkel og et tilhørende objekt. Vi kan dermed hente posisjonen til et objekt med konstant-tid søk fra hashmapet, og hente hvilket objekt som er i en posisjon fra gridet i konstant-tid.
Hvis vi largret posisjonene i hvert objekt og ikke benyttet hashmap ville vi ikke hatt samme relasjonen mellom objekt og grid. Hvor objektet er plassert i gridet sier hvilken posisjon det har. Om vi i tillegg hadde lagret posisjonen i objektetene selv hadde vi hatt posisjonene dobbelt opp.